SimTester: A Controllable and Observable Testing Framework for Embedded Systems

Tingting Yu, Witawas Srisa-an, and Gregg Rothermel

Department of Computer Science and Engineering
University of Nebraska-Lincoln
256 Avery Hall
Lincoln, NE 68588-0115
{tyu,witty,grother}@cse.unl.edu

Abstract

In software for embedded systems, the frequent use of interrupts for timing, sensing, and I/O processing can cause concurrency faults to occur due to interactions between applications, device drivers, and interrupt handlers. This type of fault is considered by many practitioners to be among the most difficult to detect, isolate, and correct, in part because it can be sensitive to execution interleavings and often occurs without leaving any observable incorrect output. As such, commonly used testing techniques that inspect program outputs to detect failures are often ineffective at detecting them. To test for these concurrency faults, test engineers need to be able to *control interleavings* so that they are deterministic. Furthermore, they also need to be able to *observe faults as they occur* instead of relying on observable incorrect outputs. In this paper, we introduce *SimTester*, a framework that allows engineers to effectively test for subtle and non-deterministic concurrency faults by providing them with greater *controllability* and *observability*. We implemented our framework on a commercial virtual platform that is widely used to support hardware/software co-designs to promote ease of adoption. We then evaluated its effectiveness by using it to test for data races and deadlocks. The result shows that our framework can be effective and efficient at detecting these faults.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging—Testing tools, Tracing; D.3.4 [Programming Languages]: Processor—Run-time environments

General Terms Reliability, Experimentation, Languages

Keywords Testing, Concurrency, Kernels, Device Drivers

1. Introduction

Concurrency faults such as data races, atomicity violations, and deadlocks are difficult to detect, isolate, and correct. This is because these faults are sensitive to execution interleavings; therefore, they often occur intermittently and unpredictably. Furthermore, certain classes of concurrency faults such as data races and atomicity violations do not always produce visible failures; therefore, techniques that inspect program outputs to detect failures are often ineffective at detecting them. As such, we have seen many instances of concurrency faults that remain dormant during testing and debugging periods and then appear during deployment [3, 10, 34].

In software for embedded systems, the frequent use of interrupts for timing, sensing, and I/O processing can cause concurrency faults to occur due to interactions between applications and interrupt handlers. As an example, occurrences of data races between interrupt handlers and applications have been reported in a previous release of *uCLinux*, a Linux OS designed for real-time embedded systems. In this particular case the serial communication line can be shared by an application through a device driver and an interrupt handler. In common instances, the execution of both the driver and the handler would be correct. However, in an exceptional operating scenario, the driver would execute a rarely executed path. If an interrupt occurs at that particular time, simultaneous transmissions of data is possible. (Section 3 provides further details).

Traditional techniques for testing for such concurrency faults require that engineers repeatedly execute a program. It is hoped that during such testing, a particular execution interleaving can reveal faults. There are two major problems with this approach. First, to reveal faults, tests must produce *observable* incorrect outputs. The absence of observable incorrect outputs, however, does not mean that there are no concurrency faults in the program. In the example described above, when data races occur, there is no guarantee that incorrect outputs are generated (e.g., in one scenario the interrupt handler wins the race and completely overwrites data written by the application). Therefore, for engineers to effectively test for data races in this case, they must be able to *observe the simultaneous transmissions and not just rely on the presence of incorrect outputs*.

As a second requirement for revealing faults, engineers must be able to exercise specific execution interleavings. However, engineers often do not have *control* over execution sufficient to exercise interleavings that are likely to expose faults. In the examples we describe in Section 3, engineers need to be able to *force an interrupt to occur at a particular location of a program*. Unfortunately, existing approaches for randomly forcing interrupts [38] are not powerful enough to support such a precise requirement. In the ideal case in which randomly invoking interrupts does expose faults, it may miss faults that can occur due to other interleavings.

There are existing techniques that can *precisely* detect concurrency faults such as data races as they occur instead of waiting to analyze the output [7, 20]. However, many of these techniques require significant source code modifications to record access sequences, monitor lock usage and track data reads and writes. Major code modifications make such approaches expensive; i.e., execution slowdown can be as high as a factor of 12. Moreover, extensive

VEE'12, March 3–4, 2012, London, England, UK.

code modifications make adopting these approaches in real-world software development environments challenging.

To achieve greater execution control, several existing approaches have tried to abstract away scheduling non-determinism. For example, there are techniques that perform dynamic analysis to permute execution interleavings [13]. There are also approaches based on model checking that can exhaustively explore execution interleavings [35]. Last, there are approaches that tightly control scheduling orders to make execution more deterministic [11]. One characteristic that these approaches have in common is that they require additional tools and runtime support that are not commonly used by real-world practitioners. As such, they have not been widely adopted by the software development community.

A further drawback with the foregoing techniques is that they have rarely been adapted to work in scenarios in which concurrency faults occur due to asynchronous interrupts. It is unclear whether these approaches can work in such a scenario for two reasons. First, controlling interrupts requires *fine-grained* execution control; that is, it must be possible to control execution at the machine code level and not at the program statement level, which is the granularity at which many existing techniques operate [13, 41]. Second, occurrences of interrupts are highly dependent on hardware states; that is, interrupts can occur only when hardware components are in certain states. Existing techniques are often not cognizant of hardware states [24, 38].

In this paper, we introduce a testing framework, *SimTester*, that provides *observability* and *fine-grained controllability* features sufficient to allow test engineers to detect concurrency faults that occur due to interactions between software and hardware. SimTester takes advantage of many features readily available in many virtual platforms to tackle the challenges of testing for concurrency errors in embedded software. Particularly, we can achieve the level of observability and controllability needed to test such systems by utilizing the virtual platform's abilities to interrupt execution without affecting the states of the virtualized system, to monitor function calls, variable values and system states, and to manipulate memory and buses directly to force events such as interrupts and traps. As such, SimTester is able to stop execution at a point of interest and force a traditionally non-deterministic event to occur. Our system then monitors the effects of the event on the system and determines whether there are any anomalies.

As stated earlier, many existing approaches for detecting concurrency faults are not widely used because they require significant deployment efforts. We have designed SimTester to overcome deployment obstacles by implementing it on a commercial virtual platform called Simics [18, 23, 45]. We chose *Simics* for several reasons. First, similar to other full-system simulators, *Simics* provides functional and behavioral characteristics similar to those of the *target hardware system*, enabling software components to be developed, verified, and tested as if they are executing on the actual systems. Second, through a rich set of *Simics* APIs, software engineers have the ability to non-intrusively observe and control various system behaviors without ever needing the source code. Third, due to its powerful device modeling infrastructure, *Simics* already plays a critical role in hardware/software (HW/SW) co-designs; therefore, adding the proposed capabilities to it will enable adoption without requiring much effort [43]. Thus, we envision that SimTester will allow several aspects of product integration testing to be moved up to the co-design phase of system development. Fourth, licensing of *Simics* is free for academic institutions, making it a good platform for research.

SimTester is implemented for applications running on x86/Linux environments. There are four major components that interact with Simics:

- A **configuration repository** stores initialization scripts containing information that includes execution break-points and variable locations that must be observed.
- An **execution controller** is an external module that can be attached to Simics. It invokes callback functions when events of interest occur (e.g., interrupts, memory read/write operations).
- An **execution observer** is another external module that can be attached to Simics. It monitors information generated by the execution controller and then records it in a file.
- An **oracle repository** stores test oracle files in the form of property requirements.[1] For example, the oracle can specify in which condition data races or deadlocks occur. Each log file is compared against an oracle file to detect a particular type of anomalous execution behavior.

By using SimTester, engineers can directly observe races and causes of deadlocks as they occur. They can also precisely control the occurrences of interrupts so that they can test every variable that can be accessed by both the application and interrupt handler for vulnerabilities to concurrency faults. SimTester yields precise detection of data races; that is, it produces no false positives. It is also effective; that is, if races or deadlocks are possible on a shared variable under test, they can be found more easily than with testing approaches that do not incorporate our framework. To evaluate the potential usefulness of SimTester we apply it to test for two classes of errors. These include data races (using an approach similar to that introduced by Higashi et al. [24] but with additional optimization) and deadlocks between device drivers and interrupt handlers. Our results show that SimTester can be effective and efficient.

The remainder of this paper is organized as follows. Section 2 provides an overview of SimTester, and Section 3 provides further details and a description of the approach by which engineers use it. Section 4 describes our evaluation of SimTester, and Section 5 provides details on related work. Finally, Section 6 concludes.

2. Introducing *SimTester*

Figure 1 depicts the overall architecture of the SimTester framework. There are four major components in the framework in addition to Simics itself. As stated earlier, Simics provides APIs that can be accessed via Python scripts; thus, all components except the test oracles are Python scripts.

The first component is the *configuration* script, the content of which includes information such as locations at which to set execution breakpoints, addresses of variables that need to be monitored, and machine instructions that need to be monitored.

The second component is the *execution controller* program. This program specifies certain events to be invoked at particular points in executions. It can generate data that can cause the system to take different execution paths. As an example, the framework can artificially create I/O interrupts simply by writing data to I/O bus or memory locations that have been mapped to hardware devices. It can also force the system to execute a particular exception handling routine by artificially creating that exception. Finally, it can force the system to execute a particular path by specifically setting a conditional value.

The third component is the *execution observer* program, which monitors and generates information that can either be recorded into a file for offline analyses (used to detect data races) or fed directly into the test oracles for online analyses (used to detect deadlocks). Any anomalies are reported in the *result file*.

[1] In the testing literature, a *test oracle* is the device by which engineers determine whether a test has elicited a failure in a system.

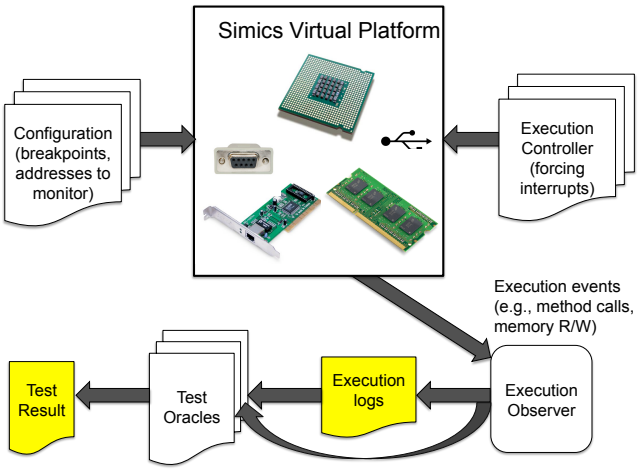

Figure 1. Overview of the SimTester architecture.

To use the framework, a test engineer first configures Simics to model the system to be tested. The engineer then writes a configuration script to set up breakpoints and a variable watch list, and specific instructions to be monitored (e.g., procedure calls and return instructions). In addition, the engineer also writes a script to specify actions to be taken when monitored events occur (Simics refers to these actions as *handlers*.) As an example, a handler for reaching an execution breakpoint could be firing a timer interrupt. Another possible handler for writing to a monitored address could make the written data available for logging or on-line analysis. Any generated information is processed by the execution observer.

In addition to the components illustrated in Figure 1, a test driver is also needed to automate the testing process. Typically, engineers conduct testing by running test programs on a system. A test driver is a program that automates the process of running test programs in a suite and managing the generated log files.

After a test driver executes a test case an event log file is generated. Log files can then be analyzed to detect anomalies, or saved for further off-line analysis. Typically, in a testing process that detect faults by analyzing a program's output, the test oracle is the correct output of the program. On the other hand, if a testing process detects faults by analyzing execution behaviors, then the test oracle can consist of properties that specify correct execution behaviors [47]. In this paper, we compare both types of oracles.

3. Utilizing *SimTester*

In this section, we describe and illustrate how *SimTester* can be used to test for two significant classes of concurrency faults: data races and deadlocks. These two classes of faults have been identified as the most "nasty" faults to test for in embedded software [17]. When we conduct testing for data races, the components under test include the main application, the UART device driver, and the ISRs that are associated with specified serial ports. The focus of our illustration is testing for races that occur when the application coupled with the device driver interact with an ISR.

Note that in the illustration that we present, interrupts are not nested, but our algorithms do also support nested interrupts. Also note that our illustration considers only a single ISR but our algorithm can be generically applied to handle multiple ISRs; however, in that case it is more difficult to isolate events related to any one particular ISR. In addition, because our framework forces interrupts to occur, we would like to distinguish between interrupts that occur naturally as part of program execution and forced interrupts issued by the execution control module of *SimTester*. We refer to the former type of interrupts as *self-generated*, and to the latter as *controlled*.

3.1 Data Race Detection

In prior releases of *uCLinux* version 2.4, there is a particular data race that occurs between the UART driver program `uart_start` and the UART ISR `serial8250_interrupt`. As it turns out, a similar error also existed in the early version of the 2.6 Linux kernel [26]. We provide the code snippets that illustrate the error in Figure 2.

The error can be summarized as follows. Under a normal operating condition, the ISR is always responsible for transmitting data. Routine `serial8250_startup` is responsible for initializing the UART port and assigning the ISR. To ensure that an ISR is assigned correctly, it issues an interrupt and monitors the response from the ISR. Several sources have shown that problems such as races with other processors on the system or intermittent port problems can cause the response from the ISR to get lost or cause a failure to correctly install the ISR, respectively [26, 27]. When that happens, the port is registered as "buggy" (line 5 in Figure 2) and workaround code based on polling instead of using interrupts is used (lines 8-11). Unfortunately, interrupts are not disabled in the workaround code region so by the time the workaround code is executed, it is possible for both the ISR and the workaround code to be transmitting or receiving data through the same serial port at the same time. As such, a race in this illustration occurs when:

1. the program is preempted by the ISR *immediately* after a shared memory access before it can proceed to next instruction;
2. the ISR manipulates the content of this shared memory.

Higashi et al. [24] introduce an approach to test for this fault by controlling invocations of interrupts. In that work, they used an ARM-based processor simulator and modified version of *uCLinux* with the same fault so that it could run on that simulator. Their

```
static int serial8250_startup(...)
1.  { ...
2.  if (lsr & UART_LSR_TEMT &&
3.          iir & UART_IIR_NO_INT) {
4.    if (!(up->bugs & UART_BUG_TXEN)) {
5.       up->bugs |= UART_BUG_TXEN;
6. }

static void serial8250_start_tx(...)
7.  { ...
8.  if (up->bugs & UART_BUG_TXEN) {
9.    ...
10.    transmit_chars(up);
11. } ...
}
```

Figure 2. Faulty code that can cause data races in the UART driver in Linux.

modifications included porting the code over from PPC to ARM and removal of irrelevant code to reduce the simulation time. Their methodology was to invoke an interrupt at every memory read and write operation.

For this illustration, we replicated the fault found in Fedora Core 2.6.15. We also recreated a similar testing system based on *SimTester* with two additional optimization techniques. In the first optimization technique, we apply static program analysis to detect the resources that can be affected by the UART driver and the ISR. With this optimization, we invoke interrupts only when these shared resources are accessed. Second, we also check the system states to ensure that it is possible to invoke interrupts when those resources are accessed. These two optimizations should significantly reduce the time required to conduct testing. Next, we discuss the configuration of *SimTester* that allows engineers to test for this fault.

Configuration of the Test System

To test for data races we need to provide two components to the system under test: the test input and conditions governing when to invoke interrupts from within the system. In this case, a *test case* is used as the test input for the program under test (PuT) (which in this case includes the application and any device driver running under non-interrupt service routine context that is called by the application.) In the case we are considering, the PuT includes an application that interacts with a serial port and the UART driver. We refer to the interrupt service routine for the tested UART port as the ISR. Note that test cases for the PuT can be generated based on various criteria. We discuss the criteria we use in the next section.

Next, we need to describe each *interrupt condition* (*IC*). We express IC as a tuple: $< loc, pin >$. The first element loc specifies a code location at which to invoke an interrupt. The second element pin specifies an *Interrupt Request* (*IRQ*) line number at which to invoke the interrupt. This is needed because typically, an interrupt service routine can be associated with multiple IRQ lines. ICs are used only when the controllability module is enabled.

Observer Module. We configure the observer module so that it records runtime information that can be used by test engineers to perform off-line analysis for races. In this example, the generated information includes:

- when functions of the PuT and ISR execute and when they return, and
- when SVs are accessed by the PuT and written by the ISR.

As such, one of our configuration tasks is to set execution breakpoints in Simics to detect when functions in the PuT and ISR execute and when SVs are accessed. The algorithms to accomplish this task are provided next.

---

procedure *BasicConfig()*
1: **begin**
2: **for** each function f in the PuT and ISR
3: set execution *breakpoint* at entry point $function\_addr$ of f
4: **endfor**
5: set execution *breakpoint* on function return instruction ret
6: set execution *breakpoint* on interrupt return instruction $iretd$
7: **end**

---

procedure *RaceObserver()*
require: procedure *BasicConfig()*
1: **begin**
2: **for** each SV in list l_{SV}
3: set memory read/write *breakpoint* at SV
4: **endfor**

5: **switch** ($breakpoint$)
6: **case** *function_addr*:
7: func_list.push({$func\_addr$, ebp, esp})
8: **if** $func\_addr$ == ISR_entry
9: log "ISR_entry"
10: is_ISR = $true$
11: **endif**
12: **case** *ret*:
13: **if** esp == func_list.top[$index\_esp$]
14: **if** func_list.top[$index\_func$] == ISR_entry
15: is_ISR = $false$
16: log "ISR_exit"
17: **endif**
18: func_list.pop()
19: **endif**
20: **case** *iretd*:
21: **if** ebp == ebp_switch
22: log "iretd"
23: /*log program counter in next instruction*/
24: $next\_pc()$
25: **endif**
26: **case** SV:
27: /*check if SV is accessed by the PuT*/
28: **if** ebp == func_list.top[$index\_ebp$]
29: **if** is_ISR == $false$
30: log "PuT", SV,SV_{access}, pc
31: /*save Reg[ebp] content*/
32: ebp_switch = ebp
33: **else** /*interrupt handler context*/
34: **if** SV_{access} == $write$
35: log "ISR", SV, SV_{access}
36: **endif**
37: **endif**
38: **endif**
39: **end**

Execution memory breakpoints are set in *BasicConfig*, including function entry addresses $function\_addr$ (line 3), function return instructions ret in the PuT and ISR (line 5), and interrupt return instructions $iretd$ (line 6). Races occur when both the PuT and ISR access the same memory location, so a memory breakpoint for

```
...
PuT, $xmit->tail$, read, pc1
ISR entry
ISR, $xmit->tail$, write
ISR exit
IRETD
pc1+1
...
```

Figure 3. Sample trace information for race detection.

each SV address is set in callback function *RaceObserver*, which is invoked whenever the execution reaches a breakpoint.

One challenge in setting up a breakpoint for each SV is that we need to be able to obtain the dynamic address of that SV. One option for doing this is to parse the symbol table. However, the symbol table provides only global variable addresses so our system may miss other shared variables such as local pointers. To obtain the address of each SV that has been identified by our static analysis (Section 4), we first create an instrumented version of the ISR so that it dynamically prints each SV address. We then iteratively inject data into the device port or adjust the device's register so that we can obtain the addresses of all SVs in the ISR stored in l_{SV}. Our function *RaceObserver* sets breakpoints at these shared variable addresses in line 3 of *RaceObserver*.

To isolate the PuT and ISR from other applications in the system or other ISR invocations, we statically identify all function names in the PuT and their entry addresses. We also identify the entry address to the ISR. These addresses can be obtained by parsing the symbol tables. Furthermore, we monitor the function return instruction (*ret* in X86) to determine whether a function or the ISR has returned, and the interrupt return instruction (*iretd* in X86) to determine whether the PuT has been recovered from the interrupt context.

At runtime, we keep a *calling stack* named $func\_list$. When a function or an ISR is invoked its <address, frame pointer, stack pointer> is added to *calling stack* (line 7). When a shared variable is accessed, we compare the current frame pointer ebp with the frame pointer on top of *calling stack* (line 28). This mechanism allows us to ignore those shared variables that might be accessed by a different ISR or a different program on the same ISR. If a ret instruction is encountered, by comparing the current stack pointer esp with the stack pointer on top of *calling stack* (line 13), we can determine whether the current function or the ISR has returned.

A function is popped from *calling stack* if its ret instruction is reached. Program counter pc is recorded twice to determine whether the PuT is actually preempted between a shared variable access and its following instruction. The first time is when a shared variable is accessed (line 30) under a non-interrupt service routine context, and the second time is after an interrupt returns (line 24). An interrupt return instruction $iretd$ is recorded to indicate termination of an interrupt context. Note that the mere presence of an $iretd$ does not imply that an interrupt will jump back to the PuT, because more than one device can issue interrupts and call $iretd$ instructions. To overcome this problem, an $iretd$ is logged only when its frame pointer is equal to the frame pointer when a shared variable is accessed in the PuT (line 22).

In summary, events logged for testing race conditions include: (1) read/write accesses to shared variables (SV_{access} by the PuT); (2) entry to the ISR; (3) a write to an SV by the ISR; (4) return from the ISR; (5) context switches from the ISR to the PuT. Figure 3 illustrates a sample of trace information recording these events for this example.

Note that there is a race in the trace given in Figure 3. By observing the program counter when SV is accessed by the PuT and the interrupt recovery point, we can determine that an interrupt occurs right after $xmit- > tail$ is read by the PuT.

Controller Module. When engineers enable the controller module *RaceController*, a *controlled* interrupt is invoked right after a shared variable access by the PuT. Simics provides a $simple\_interface$ API to issue an interrupt on a specific IRQ line. This interface guarantees that the interrupt happens before the subsequent instruction. As such, when our test system reaches a memory breakpoint, the observer module is called. If the controller module is enabled, the observer module tries to invoke an interrupt right after the access to SV.

It is not always realistic, however, to invoke an interrupt whenever we want. For example, the interrupt enable register and possibly other control registers have to be set to enable interrupts. Even if interrupts are enabled, they can be temporarily disabled. The following routine is the routine in *RaceController* used to determine whether it is possible to issue an interrupt.

```
procedure ISR_enabled(int p)
/*p is the pin number for a certain interrupt*/
1: begin
2: if eflags[9] != 0 and ioapic.redirection[p] == 0
     and ioapic.pin_raised[p] == LOW:
3:     return true
4: else
5:     return false
6: endif
7: end
```

There are two general steps that our system takes prior to invoking a *controlled interrupt*. First, the controller module checks the status of the local interrupt and global interrupt bits to see if interrupts are enabled. In an X86 architecture the global interrupt bit is the ninth bit of the eflags register (line 2 of *ISR_enabled*). When this bit is set to 1, the global interrupt is disabled, otherwise it is enabled. For local interrupts, Simics uses the Advanced Programmable Interrupt Controller (APIC) as its interrupt controller. As such, our system checks whether the bit controlling the UART device is masked or not. Our system also checks whether a *self-generated* interrupt is about to be issued by examining the current pin status. If this is true, the *controlled* interrupt will not be invoked.

Second, our system invokes only one *controlled* interrupt per test run. This is done to avoid *fault masking* effects, which may occur in cases where multiple interrupts fire and cause a failure that would be evident in the presence of a single interrupt to be "masked" by the presence of the second. Thus, our system needs to first check a flag to determine whether a controlled interrupt has already been invoked in the current run. If it has, the test system does not monitor any further events in this run. Once it has been determined that there has not been any invocation of a controlled interrupt in this run, the system then checks to see whether the last accessed SV has already been tested in prior runs. If it has not, the system enables the control register for UART and then calls the $simple\_interrupt$ API.

Note that, given the foregoing approach, there can be multiple runs of each test case, and the number of runs depends on the number of SVs that must be tested. With the controller module disabled, the PuT runs $|tc|$ times during a testing process, where $|tc|$ is the number of test cases. With the controller module enabled, the number of runs is $|tc| * (|int| + 1)$, where $|tc|$ is the number of test cases and $|int|$ is the number of controlled interrupts issued. We also need to run the PuT one additional time for each test case to determine whether all SVs have been accessed.

```
void xhci_watchdog(...)
1:  {
2:        xhci = ep->xhci;
3:         ...
4:        spin_lock(&xhci->lock);
5:        ...
6:        spin_unlock(&xhci->lock);
7:  }
irqreturn_t interrupt(int irq, void *dev_id)
8:  {
9:        xhci = get_dev(dev_id);
10:       ...
11:       spin_lock(&xhci->lock);
12:       ...
13: }
```

Figure 4. Faulty code that can cause deadlocks.

3.2 Deadlock Detection

It is common for interrupt handlers to use non-preemptive mechanisms such as spin-locks instead of preemptive mechanisms such as semaphores to protect critical code regions. This is because typically, ISR code is not allowed to sleep. As such, interrupt code is short and deterministic so the amount of time that a task must wait to enter the critical region should be predictably short [15]. That said, there are many real world examples including the one used in this illustration that show how incorrect usage of spinlocking mechanisms can cause priority inversions and ultimately deadlocks between the PuT and ISR. In the example of Figure 4, deadlocks occur because:

- the PuT is preempted by the ISR while the PuT is holding a spin lock,
- the ISR tries to acquire the same spinlock and becomes stuck in the spinlock loop.

Because the ISR has higher priority, it can become stuck in this loop. To test for such deadlocks, we configure our test system in a manner similar to that used to test for races. However, test cases for the system are generated to adequately cover $spin\_lock$ and $spin\_unlock$ pairs in the PuT instead of shared variables. In this illustration, we assume that $spin\_lock$ and $spin\_unlock$ are properly paired. Next, we describe the configurations of the execution observer and execution controller.

Configuration of the Test System

In a uniprocessor environment, an observer does not increase fault detection effectiveness because deadlocks can be easily observed (e.g., systems hang or show no responses). However, the proposed test system can still be useful in informing testers of the sources of deadlocks. Furthermore, it can support deadlock detection in multiprocessor systems where deadlock might cause one or more processors to stop making execution progress but other processors continue to do useful work.

Observer Module. We configure the observer module so that it records runtime information that can be used to both issue a runtime warning and log for offline analysis. In this deadlock example, the generated information includes:

- when functions of the PuT and ISR execute and return;
- when a spinlock is acquired by the PuT and by ISR, and
- when a spinlock is released by the PuT.

As in the above example, when the spinlock is acquired by the the PuT, the observer sets the lock condition to true. If the ISR tries to obtain the same lock, the observer first checks whether the lock condition is true, and then compares the requested lock with that held by the PuT. If they are the same, we have a deadlock. In this case, the observer issues a warning once the deadlock is detected. In both environments, it also records the event so that engineers can perform analysis offline. Function *DeadlockObserver* describes our algorithm to detect deadlocks.

procedure *DeadlockObserver()*
require: procedure *BasicConfig()*
1: **begin**
2: set execution *breakpoint* on entry point of $spin\_lock$ and $spin\_unlock$

3: **switch** ($breakpoint$)
4: **case** *func_addr*:
5: func_list.push({$func\_addr, ebp, esp, lock\_obj$})
6: **if** func_addr == ISR_entry
7: is_ISR = $true$
8: **endif**
9: **case** *spin_lock*:
10: **if** ebp == func_list.top[$index\_ebp$]
11: **if** is_ISR and is_lock and eax == lock_obj
12: print"deadlock occurs"
13: **else**
14: func_list.push({$func\_address, ebp, esp, eax$})
15: **endif**
16: **endif**
17: **case** *spin_unlock*:
18: **if** ebp == func_list.top[$index\_ebp$]
19: func_list.push({$func\_address, ebp, esp, eax$})
20: **endif**
21: **case** *ret*:
22: **if** esp == func_list.top[$index\_esp$]
23: **if** func_list.top[$index\_func$] == ISR_entry
24: is_ISR = $false$
25: **endif**
26: **if** fun_list.top[$index\_func$] == $spin\_lock$
27: /*spinlock returns*/
28: **if** !is_ISR
29: is_lock = $true$
30: lock_obj = func_list.top[$index\_eax$]
31: **endif**
32: **endif**
33: **if** fun_list.top[index_func] == $spin\_unlock$
34: **if** !is_ISR
35: is_lock = $false$
36: lock_obj = $null$
37: **endif**
38: **endif**
39: func_list.pop()
40: **endif**
41: **end**

Functions $spin\_lock$ and $spin\_unlock$ are commonly used by various applications. As such, we need to isolate calls to these two functions that come from the PuT and ISR. Again, we use the *calling stack* to dynamically store information on called functions during virtualization (lines 5, 14, 19).

Initially, lock_obj is set to $null$. The lock is acquired after $spin\_lock$ returns (line 29), and released after $spin\_unlock$ returns

(line 35). Besides the frame pointer and stack pointer, each function in *calling stack* carries a lock object, indicating whether the current running function is holding a lock or which lock it is holding. As such, a locked object can always be obtained by examining the top of stack (line 30).

A deadlock occurs under three conditions (line 10): (1) the ISR is executing; (2) a lock is held by the PuT; (3) the ISR is stuck in the same spinlock loop as the object of the held lock.

Deadlock detection is different from race detection in that a deadlock warning is issued instead of simply recording deadlock information. Once a deadlock is detected, the test script terminates execution and reinitializes the test system for the next test run. Because a lock object is passed as a parameter to $spin\_lock$ or $spin\_unlock$, the object can be obtained by reading the CPU eax register in the X86 architecture. Note that this is architecture and compiler dependent. However, the approach should be generalizable to other architectures and compilers as long as we know how the lock object is passed.

Controller Module. The controller module *DeadlockController* is implemented following the same steps as the controllability module for race conditions, except that an interrupt is issued after a spinlock is acquired by the PuT instead of invoking interrupts on each shared variable access.

4. Evaluation

To evaluate SimTester, we applied our approach to the UART device driver on a preemptive kernel version of Fedora Core 2.6.15. The driver includes two files, serial_core.c and 8250.c, containing 1896 and 1445 lines of non-comment code, respectively. The main application transmits character strings to and receives character strings from console via the UART port. Note that in this paper we apply our testing process only to the UART driver. However, the same process is also applicable to other types of device drivers.

Our approach requires the use of existing test cases, so we generated test cases for the system based on a code-coverage-based test adequacy criterion. To generate test cases relevant to race conditions, we first statically identified shared variables (SVs) between the PuT and ISR. We use the precise shared variable detection algorithm proposed in [29], but we are interested only in shared variables that are read by the PuT and written by the ISR, or written by both the PuT and the ISR. We label each shared variable as a "definition" or "use" through our analysis. After SVs are identified, we generate a set of test cases that cover the feasible SVs (SVs for which there exists a possible execution of the program which executes them) in the PuT. This process produced 12 test cases.

To generate test cases relevant to deadlocks, we sought to find test cases that adequately cover $spin\_lock$ and $spin\_unlock$ pairs in the PuT instead of shared variables. (In the case of our target program, we know that all occurrences of $spin\_lock$ and $spin\_unlock$ are properly paired; in practice a static analysis could initially determine this and flag unpaired occurrences for attention by the test engineer.) In the case of our target program, which contains only three spin-lock pairs, this process resulted in two test cases (one of which covered two of the pairs).

To better assess the cost and effectiveness of our approach, we considered both the approach and two alternative baseline approaches. In the discussion that follows, we refer to our approach as the *conditional controllability* approach, because it involves issuing controlled interrupts under certain conditions. The second approach that we considered, *no controllability*, involves testing the program without any controlled interrupts; this is the approach that test engineers would normally use. The third approach that we consider, *random controllability*, involves issuing controlled interrupts at random program locations after shared variable accesses and without checking interrupt conditions.

We measured execution times for the foregoing approaches by embedding a timer in the Simics module. As such, the reported times are the actual times spent by Simics to execute the program.

4.1 Testing for Race Conditions

We begin by considering the target program as given, and evaluate the effectiveness and efficiency of our race condition testing approach on that program.

We first applied conditional controllability together with observability. Under this approach, across the 12 test cases utilized, 84 controlled interrupts were applied, and for each test case, one extra run was needed to determine whether all shared variables had been accessed. Thus, 96 test runs were required to finish testing the target program with an average execution time of 77.91s per test run. Including self-generated interrupts, the number of interrupts generated for the target program was 352. In the course of applying the approach, we detected a race in function $uart\_write\_room$ of serial_core.c, which we later determined had been corrected in subsequent versions of the system. By running the system with observability turned off, we determined that this fault can be detected only with observability enabled; in other words, it is a fault that did not propagate to output on our particular test inputs.

We next tested our target program with no controllability. In this case, the only interrupts that occur are self-generated interrupts. Because runs of each given test can conceivably differ, we ran each test on the program 500 times. The total number of interrupts observed was 16,500. Over the 6000 total test runs, average execution time was 74.08s per test, only 3.83 seconds less than with controllability added. None of these test runs detected the race condition detected by our first approach, however, either with observability enabled or disabled.

Finally, we tested our target program using the random controllability approach. For each test case, we ran the target program three times more than the number of runs performed under the conditional controllability approach, on each run generating an interrupt at a randomly selected program location. The total number of test runs was 288 and the number of interrupts generated was 1044. In this case the average execution time per test case was 75.17s, only 2.74 seconds less than with controllability added. Again, the race was not detected, either with observability enabled or disabled.

One important characteristic of our technique is that checking is performed before issuing a controlled interrupt. When a shared variable is accessed in the main program, the controllability module first checks to see whether it is possible to issue an interrupt, and if not, it proceeds to the next possible location. This approach can save test runs, but at the cost of checking. To quantify the tradeoffs involved, we also applied our conditional controllability approach without the checking step enabled. Recall that with checking enabled, 96 test runs were needed to issue controlled interrupts, with an average execution time of 77.91s per test. With checking disabled, on the other hand, 1428 test runs were needed to issue controlled interrupts, with an average execution time of 75.66s per test. Clearly, the checking approach saves time overall.

A second characteristic of our technique is that interrupts are issued only after shared memory accesses, and this can be much less expensive than issuing interrupts after each memory access, which is the approach used by Higashi et al. [24]. For our target program, there are 94,941 data accesses made in the course of running the 12 test cases. If an interrupt were issued after each data access, we would need 82.6 days to finish testing the target program.

4.2 Testing for Deadlock

We next consider the target program as given, and evaluate the effectiveness and efficiency of our deadlock testing approach on that program. (In this case, because our experiment is running on a uniprocessor and observability does not increase fault detection power, we consider only the effects of controllability, not the effects of observability.)

We began by running our two deadlock test cases on the target program under conditional controllability. Under this approach, only two controlled interrupts were generated, and they detected one deadlock in function $serial8250\_handle\_port$ of 8250.c, which had been reported [30] and corrected in later versions. The average execution time was 69.88s per test run.

We next ran the target program with no controllability. For each test case, we ran the program 500 times; the 1000 tests had an average execution time of 75.24s, and 4000 self-generated interrupts were observed. However, no deadlocks were detected.

Finally, we ran the program using the random controllability approach. Here we ran the program three times as many as the number of runs performed for conditional controllability, issuing an interrupt at a random program location. The total number of controlled interrupts was six and the total number of interrupts generated was 15. The average run time was 75.24s per test. Here also, no deadlocks were detected.

Notice that the average execution times of the latter two approaches are higher than that of the conditional controllability approach. This result, at first, appears counter-intuitive. However, in our experiment with conditional controllability, we terminated the execution once a deadlock has been detected. On the other hand, the latter two approaches did not detect deadlocks, so the program ran to completion.

4.3 Fault Detection Effectiveness

While the results of the foregoing studies are encouraging, the numbers of naturally occurring faults found in the target program was low, rendering comparisons of the fault detection effectiveness of the approaches less meaningful. To further investigate fault detection effectiveness we followed a process often utilized in the software testing research community [2]; namely, the use of seeded faults.

In this case, we injected 12 potential race condition faults and 11 potential deadlock faults into 8250.c by making syntactic changes to the code. For race conditions, we removed statements corresponding to critical section protection (e.g., $spin\_lock$, $spin\_lock\_irq$). For deadlock, we changed statements corresponding to interrupt disable and enable pairs (e.g., $spin\_lock\_irq$ and $spin\_unlock\_irq$) into pure spin lock pairs (e.g., $spin\_lock$, $spin\_unlock$). Of the 23 potential faults thus created, further examination revealed that seven of the potential race condition faults, and seven of the potential deadlock faults, could not possibly be triggered on the system on its given hardware platform, so we removed those. This left us with five potentially revealable race condition faults and four potentially revealable deadlock faults.

Given the faults thus seeded, we ran our test cases on the faulty systems using conditional and random controllability, and in the case of race detection, with observability enabled and disabled. For the race condition detection approach, conditional controllability detected two of the five faults. One of these faults was detected both with and without observability. The same fault was also detected with random controllability, but only with observability enabled because in this case the fault does not propagate to output. This occurred because interrupts issued by conditional controllability visited more unprotected shared variables that can cause incorrect output, and these shared variables are not visited by random controllability.

The second fault revealed in our race detection trial was revealed not through observability, but rather, through output, for both conditional controllability and random controllability. The reason this occurred is because the fault was not actually caused by our defined race condition, but rather, by another type of atomicity violation. In particular, a read-write SV pair in the main program is supposed to be atomic, but the ISR read this SV before it was updated in the main program. This outcome shows that, while our approach does not specifically target other types of faults, it may catch them as byproducts.

We also inspected the three potential race condition faults that were not detected by any techniques. We determined that the reason for their omission was that the interrupt handler in each of the versions does not share variables or read variables with the main program. This does not mean that the code regions involved do not need to be protected, because other ISRs may share memory locations, or programmers may intentionally cause the regions to execute without interruption.

Where deadlock faults were concerned, we discovered that conditional controllability detected all four, while random controllability detected two.

5. Further Discussion

Our observer module considers one type of definition of a race condition. In practice, testers can adopt different definitions because there is not a single general definition for the class of race conditions that occur between an ISR and a PuT. As noted above, for the four faulty versions on which the ISR and the PuT do not share read-write and write-write variables, we still found one fault with controllability enabled. This fault is related to an atomicity violation, as a code region in the main program is supposed to execute atomically, e.g., before a shared variable is updated in the main program, and an interrupt occurs and the wrong data is read.

Our approach injects data into device ports and forces an interrupt handler to execute one path. The data we inject is the same as the test input given to the program. For example, if an application sends the string "hello" to the UART console passed by UART *transmitter buffer*, a controllability module would inject "world" into the UART *transmitter buffer* to force an interrupt to occur after a certain access. It is also possible to have multiple paths by which shared variables can exist in interrupt handlers. Testers can extend our method by forcing interrupt handlers to execute different paths, which may increase the probability of revealing faults.

However, no faults are left undetected due to missing shared variables or spin locks in the other paths of the ISR in our target program. It is also possible to force an interrupt handler to execute only the paths that have definition-use relationships with the main program. This may further reduce the number of controlled interrupts and test runs. To do this, the value schedule approach proposed by Chen et al. [13] could be adapted.

To force an interrupt to occur, our controllability module issues a new interrupt. However, races and deadlocks can occur relative to the interrupt generated by the target program itself. For example, suppose an interrupt is requested by device driver code, but is not immediately processed for some reason (e.g., device port delay). The interrupt handler associated with this interrupt may be executed later within a spin lock pair or after a shared variable access, and thus a race condition or a deadlock may occur. Our controllability module can be further extended to deal with such cases. For example, when an interrupt is triggered, the module can delay this interrupt by masking its interrupt enable register, and issue the interrupt after a certain event happens (e.g., shared variable access, spin lock is acquired). If there is no such event, the interrupt is issued on exiting the PuT.

In practice, when testing software components (e.g., device drivers and interrupt handlers), the first task that a test engineer must accomplish is to gain confidence that the software component is developed correctly. In our study, the analysis involves a test program, the interrupt handler that interacts with the device driver, and the device driver code. The key point here is that the tester focuses on a specific component and how it interacts with the rest of the components. If the focus changes to a different component, the same analysis can be applied to test the new component. As such, the proposed approach is not designed to test the entire system at once. Instead, it is more suitable for component testing.

In our current work, the test generation process was done manually (which is currently the norm in practice). Our study considers a test input to include input values and interrupt scheduling. However, there is no reason the approach could not also utilize input values created using existing test case generation approaches (such as dynamic symbolic execution [9, 42].) A problem with such approaches by themselves is that they generate large numbers of test cases with no methodology for judging system correctness beyond looking for crashes. Our approach provides more powerful, automated oracles, and thus should ultimately facilitate the use of larger numbers of automatically generated test cases.

6. Related Work

There are several techniques for testing embedded systems with a particular focus on interrupt related faults. As already noted, Higashi et al. [24] detect race conditions caused by interrupt handlers via a mechanism that causes interrupts to occur at all possible times. Their method and ours artificially amplify the frequency of interrupts to evaluate whether these interrupts can cause faults. However, our work has several advantages. First, by adding observability, we increase the power of fault detection while Higashi's method focuses on controllability. Second, instead of firing interrupts at all memory access instructions of a *PuT*, we issue interrupts only at shared variable accesses, which significantly reduces testing cost. Third, we adopt coverage criteria to cover all feasible shared variables in the *PuT* instead of using arbitrary inputs; this can help the program execute code regions that are more race-prone. Fourth, Higashi's method issues all interrupts during one program run. This may cause problems with fault masking and cascading errors. In contrast, we issue just one interrupt during a given program run. Fifth, Higashi's method does not consider situations in which interrupts cannot occur; however, our technique can determine whether it is possible to issue interrupts at runtime by observing hardware states, which further improves testing efficiency. Another implementation difference is that we built our framework on a virtual platform while their approach is built on an processor emulator.

In other work, Regehr et al. [38] use random testing to test Tiny OS applications. They propose a technique called restricted interrupt discipline (RID) to improve naive random testing (i.e., firing interrupts at random times) by eliminating aberrant interrupts. In our approach, however, interrupts are conditionally fired instead of randomly. Our evaluation has shown that conditionally fired interrupts increase the chances of revealing faults while reducing cost. Lai et al. [33] present a notation for modeling interrupt-driven nesC applications for testing purposes. They formulate two test adequacy criteria based on the notation. Their approach does not provide observability or controllability. In their approach, test cases are randomly selected from a test pool until coverage has been achieved. We believe that this process can be made more effective by adding controllability.

Static analysis techniques discover paths and regions in code that might be susceptible to concurrency faults (e.g., [19, 25, 46]). Techniques based on program state modeling and transitions (e.g., [16]) have been used to verify device drivers and kernels [5, 19]. There are also static analysis techniques used to verify embedded software with interrupts. Tan et al. [44] designs a type of annotation that can be used to detect OS concurrency faults related to interrupts. Their approach is based on statically analyzing comments and code. Brylow et al. [8] apply a model checker to check interrupt related real time properties such as interrupt latency. Schlich et al. [40] propose a technique called *interrupt handler reduction* to reduce the number of program locations at which an interrupt needs to be considered. The goal of this technique is to reduce program states in model checking. There are several drawbacks associated with static analysis techniques such as these. First, state explosion in static analysis may cause scalability problems. Second, static analysis can report false positives due to imprecise local information and infeasible paths. Third, as embedded systems are highly dependent on hardware, it is difficult for static analysis to annotate all operations on manipulated hardware bits; moreover, hardware events such as interrupts usually rely on several operations among different hardware bits.

Dynamic analysis tends to be more accurate than static analysis for detecting concurrency faults (see e.g., [28, 36]). Techniques that do not control thread interleavings (e.g., [39]) can miss faults that are caused by different interleavings. Moreover, existing dynamic techniques cannot leverage the scheduling policies for interrupts and main programs because such techniques do not consider priority based preemption. Controlling thread interleavings at runtime is a way to increase the chances of exposing faults. Sen et al. [41] first identify potential concurrency faults, and then control the underlying scheduler by inserting delays at context switch points. This technique, however, may not apply in our context due to different scheduling policies. (Hardware interrupts are asynchronous events generated from hardware, so they are not scheduled by kernel thread schedulers. For example, it is impossible to force shared variable accesses at the same time because an interrupt handler cannot be preempted by a normal program.) Other dynamic techniques such as interrupt tracing [22] are not designed to detect faults. Our goal is to utilize test cases to detect concurrency faults involving runtime execution of application and hardware interrupt service routine. Thus, we employ dynamic analysis.

Virtual platforms have been used to support tracing, replay, and debugging [1, 6, 12, 31, 37]; however, they have rarely been used to support testing. One notable example is work by Goh et al. [21]. They introduced a VM-based online testing approach to supplement off-line testing. With off-line testing, all possible test inputs may not be known ahead of time since embedded software systems are often influenced by external environments. Their work, however, did not consider the use of observability and controllability to enhance testing effectiveness. On the other hand, there have been some efforts to extend virtual platforms to provide greater observability in security domain [4, 14, 32]. However, these efforts did not utilize the additional observing power for testing.

7. Conclusion

The frequent use of interrupts for timing, sensing, and I/O processing in embedded software can cause concurrency faults to occur due to interactions between applications, device drivers, and interrupt handlers. This type of fault is considered by many practitioners to be among the most difficult to detect, isolate, and correct, in part because it can be sensitive to interleavings and often occurs without causing any observable incorrect output. In this work, we introduce *SimTester*, a framework that provides test engineers with the ability to precisely *control* execution events and *observe* runtime context at critical code locations. The framework is built on a commercial virtual platform that is commonly used as part of the hardware/software co-design process.

The main benefit of using virtual platforms is the ability to interrupt execution without affecting the states of the virtualized system. Furthermore, we can monitor function calls, variable values and system states, and manipulate memory and buses directly to make typically non-deterministic execution events more deterministic. To illustrate the effectiveness of SimTester, we use it to test software systems with two real-world concurrency faults, races and deadlocks, that were uncovered in the previous releases of Linux kernels. They are the result of untimely occurrences of interrupt at improperly synchronized code locations. We also apply SimTester to a number of seeded faults. Our evaluation clearly shows that our technique is effective in detecting faults that are unlikely to be detected in approaches that rely on observing incorrect outputs.

Acknowledgments

This work is supported in part by the National Science Foundation through awards CNS-0720757 and by the Air Force Office of Scientific Research through award FA9550-10-1-0406.

References

[1] L. Albertsson. Simulation-Based Debugging of Soft Real-Time Applications. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium (RTAS)*, pages 107–108, Taipei, Taiwan, 2001.

[2] J. H. Andrews, L. C. Briand, and Y. Labiche. Is Mutation An Appropriate Tool for Testing Experiments? In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 402–411, 2005.

[3] Apache Software Foundation. Datarace on org.apache. catalina.loader.webappclassloader. https://issues.apache.org/bugzilla/show_bug.cgi?id=37458.

[4] K. Asrigo, L. Litty, and D. Lie. Using VMM-based Sensors to Monitor Honeypots. In *Proceedings of the 2nd International Conference on Virtual Execution Environments (VEE)*, pages 13–23, Ottawa, Ontario, Canada, 2006.

[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. In *Proceedings of the 1st ACM SIGOPS European Conference on Computer Systems (EuroSys)*, pages 73–85, Leuven, Belgium, 2006.

[6] S. Bhansali, W.-K. Chen, S. de Jong, A. Edwards, R. Murray, M. Drinić, D. Mihočka, and J. Chau. Framework for Instruction-level Tracing and Analysis of Program Executions. In *Proceedings of the 2nd International Conference on Virtual Execution Environments (VEE)*, pages 154–163, Ottawa, Ontario, Canada, 2006.

[7] M. D. Bond, K. E. Coons, and K. S. McKinley. PACER: Proportional Detection of Data Races. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 255–268, Toronto, Ontario, Canada, 2010.

[8] D. Brylow, N. Damgaard, and J. Palsberg. Static Checking of Interrupt-driven Software. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE)*, pages 47–56, Toronto, Ontario, Canada, 2001.

[9] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX conference on Operating Systems Design and Implementation (OSDI)*, pages 209–224, San Diego, California, 2008.

[10] Canonical Ltd. Launchpad: data-races-implementation sql crash. https://bugs.launchpad.net/f-4d-cb/+bug/516622.

[11] R. H. Carver and K.-C. Tai. Replay and Testing for Concurrent Programs. *IEEE Software*, 8:66–74, March 1991.

[12] M. Chang, E. Smith, R. Reitmaier, M. Bebenita, A. Gal, C. Wimmer, B. Eich, and M. Franz. Tracing for Web 3.0: Trace Compilation for the Next Generation Web Applications. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, pages 71–80, Washington, DC, USA, 2009.

[13] J. Chen and S. MacDonald. Testing Concurrent Programs Using Value Schedules. In *Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 313–322, Atlanta, Georgia, USA, 2007.

[14] J. Chow, T. Garfinkel, and P. M. Chen. Decoupling dynamic program analysis from execution in virtual environments. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, 2008.

[15] J. Corbet, A. Rubini, and G. Kroah-Hartman. *Linux Device Drivers, 3rd Edition*. O'Reilly and Associates, 2005.

[16] M. B. Dwyer and L. A. Clarke. Data Flow Analysis for Verifying Properties of Concurrent Programs. In *Proceedings of the 2nd ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE)*, pages 62–75, New Orleans, Louisiana, 1994.

[17] EETIMES. Five top causes of nasty bugs. Web page. http://www.eetimes.com/design/embedded/4008917/Five-top-causes-of-nasty-embedded-software-bugs.

[18] J. Engblom. Virtutech device modeling language. White-Paper, 2009. http://www.virtutech.com/files/whitepapers/wp-dml-2009-03-30-letter.pdf.

[19] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking System Rules Using System-specific, Programmer-written Compiler Extensions. In *Proceedings of the 4th Symposium on Operating System Design & Implementation (OSDI)*, pages 1–16, San Diego, California, 2000.

[20] 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 (PLDI)*, pages 121–133, Dublin, Ireland, 2009.

[21] O. Goh and Y.-H. Lee. Schedulable Online Testing Framework for Real-time Embedded Applications in VM. In *Proceedings of the 2007 International Conference on Embedded and Ubiquitous Computing (EUC)*, pages 730–741, Taipei, Taiwan, 2007.

[22] G. Gracioli and S. Fischmeister. Tracing Interrupts in Embedded Software. In *Proceedings of the 2009 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 137–146, Dublin, Ireland, 2009.

[23] C. Haggstrom. *Detection of Memory Allocation Bugs on System Level*. PhD thesis, Umea University, Sweden, 2005. Adviser-Mikael Rannar.

[24] M. Higashi, T. Yamamoto, Y. Hayase, T. Ishio, and K. Inoue. An Effective Method to Control Interrupt Handler for Data Race Detection. In *Proceedings of the 5th Workshop on Automation of Software Test (AST)*, pages 79–86, Cape Town, South Africa, 2010.

[25] D. Hovemeyer and W. Pugh. Finding Concurrency Bugs in Java. In *In Proceedings of the PODC Workshop on Concurrency and Synchronization in Java Programs*, Newfoundland, Canada, 2004.

[26] I. Jackson. IRQ handling race and spurious IIR read in 8250.c. Web page. https://lkml.org/lkml/2009/3/12/379.

[27] I. van Lil. Serial: UART_BUG_TXEN race conditions. Web page. http://lkml.org/lkml/2006/6/28/81.

[28] P. Joshi, M. Naik, C.-S. Park, and K. Sen. CalFuzzer: An Extensible Active Testing Framework for Concurrent Programs. In *Proceedings of the 21st International Conference on Computer Aided Verification (CAV)*, pages 675–681, Grenoble, France, 2009.

[29] V. Kahlon, Y. Yang, S. Sankaranarayanan, and A. Gupta. Fast and Accurate Static Data-race Detection for Concurrent Programs. In *Proceedings of the 19th International Conference on Computer Aided Verification (CAV)*, pages 226–239, Berlin, Germany, 2007.

[30] Kernel Trap. deadlock in 3c59x driver. Web page. http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=commit;h=4bf3631cdb012591667ab927fcd7719d92837833.

[31] T. Koju, S. Takada, and N. Doi. An Efficient and Generic Reversible Debugger using the Virtual Machine Based Approach. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments (VEE)*, pages 79–88, Chicago, IL, 2005.

[32] K. Kourai and S. Chiba. HyperSpector: Virtual Distributed Monitoring Environments for Secure Intrusion Detection. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments (VEE)*, pages 197–207, Chicago, IL, 2005.

[33] Z. Lai, S. C. Cheung, and W. K. Chan. Inter-context Control-flow and Data-flow Test Adequacy Criteria for nesC Applications. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, pages 94–104, Atlanta, Georgia, 2008.

[34] Liang T. Chen. The Challenge of Race Conditions in Parallel Programming. http://developers.sun.com/solaris/articles/raceconditions.html.

[35] O. Lichtenstein and A. Pnueli. Checking that Finite State Concurrent Programs Satisfy their Linear Specification. In *Proceedings of the ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 97–107, New Orleans, Louisiana, 1985.

[36] C.-S. Park and K. Sen. Randomized Active Atomicity Violation Detection in Concurrent Programs. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 135–145, Atlanta, Georgia, 2008.

[37] A. Pohle, B. Döbel, M. Roitzsch, and H. Härtig. Capability Wrangling Made Easy: Debugging on a Microkernel with Valgrind. In *Proceedings of the 6th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, pages 3–12, Pittsburgh, Pennsylvania, USA, 2010.

[38] J. Regehr. Random Testing of Interrupt-driven Software. In *Proceedings of the 5th ACM International Conference on Embedded Software (EMSOFT)*, pages 290–298, Jersey City, NJ, USA, 2005.

[39] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[40] B. Schlich, T. Noll, J. Brauer, and L. Brutschy. Reduction of Interrupt Handler Executions for Model Checking Embedded Software. In *Proceedings of the 5th international Haifa Verification Conference on Hardware and Software: Verification and Testing (HVC)*, pages 5–20, Haifa, Israel, 2011.

[41] K. Sen. Race Directed Random Testing of Concurrent Programs. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 11–21, Tucson, AZ, USA, 2008.

[42] K. Sen, D. Marinov, and G. Agha. CUTE: A Concolic Unit Testing Engine for C. In *Proceedings of the 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 263–272, Lisbon, Portugal, 2005.

[43] J. Takalo, J. Kaariainen, P. Parviainen, and T. Ihme. Challenges of Software-Hardware Codesign. White Paper, 2007. http://www.vtt.fi/inf/pdf/workingpapers/2008/W91.pdf.

[44] L. Tan, Y. Zhou, and Y. Padioleau. aComment: Mining Annotations from Comments and Code to Detect Interrupt Related Concurrency Bugs. In *Proceeding of the 33rd International Conference on Software Engineering (ICSE)*, pages 11–20, Waikiki, Honolulu, HI, USA, 2011.

[45] Wind River. Wind River Simics. Web-page, 2011. http://www.simics.net.

[46] J. W. Voung, R. Jhala, and S. Lerner. RELAY: Static Race Detection on Millions of Lines of Code. In *Proceedings of the the 6th joint meeting of the European Software Engineering Conference (ESEC) and the ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE)*, pages 205–214, Dubrovnik, Croatia, 2007.

[47] T. Yu, A. Sung, W. Srisa-an, and G. Rothermel. Using Property-based Oracles when Testing Embedded System Applications. In *Proceedings of the Fourth International Conference on Software Testing, Verification and Validation (ICST)*, pages 100–109, Berlin, Germany, March 2011.