Fault Injection Test Harness

a tool for validating driver robustness

Louis Zhuang
Intel Corp.
louis.zhuang@intel.com,
louis.zhuang@acm.org

Stanley Wang
Intel Corp.
stanley.wang@intel.com

Kevin Gao
Intel Corp.
kevin.gao@intel.com

Abstract

FITH (Fault Injection Test Harness) is a tool for validating driver robustness. Without changing existing code, it can intercept arbitrary MMIO/PIO access and IRQ handler in driver.

Firstly I'll first list the requirements and design for Fault Injection. Next, we discuss a couple of new generally useful implementation in FITH

1. KMMIO - the ability to dynamically hook into arbitrary MMIO operations.

2. KIRQ - the ability to hook into an arbitrary IRQ handler,

Then I'll demonstrate how the FITH can help developers to trace and identify tricky issues in their driver. Performance benchmark is also provided to show our efforts in minimizing the impact to system performance. At last, I'll elaborate on current and future efforts and conclude.

1 Introduction

High-availability (HA) systems must respond gracefully to fault conditions and remain operational during unexpected software and hardware failures. Each layer of the software stack of a HA system must be fault tolerant, producing acceptable output or results when encountering system, software or hardware faults, including faults that theoretically should not occur. An empirical study [2] shows that 60-70% of kernel space defects can be attributed to device driver software. Some defect conditions (such as hardware failure, system resource shortages, and so forth) seldom happen, however, it is difficult to simulate and reproduce without special assistant hardware, such as an In-Circuit Emulator. In these situations, it is difficult to predict what would happen should such a fault occur at some time in the future. Consequently, device drivers that are highly available or hardened are designed to minimize the impact of failures to a system's overall functionality.

Developing hardened drivers requires employing fault avoidance software development techniques early in the development phase. To

eliminate faults during development and confirm a driver's level of hardening, a developer can test a device driver by injecting or simulating fault events or conditions. The focus of this paper is on the injection or simulation of hardware faults. Injection of software faults will be considered in future version.

FITH simulates hardware-induced software errors without modifying the original driver. It offers flexible customization of hardware fault simulation as well as provides command-line tool for facilitating test development. FITH can also provide the ability to log the route of an injected fault, thereby enabling driver developers to diagnose the tested driver.

2 Requirements

This section describes some requirements for FITH; we derived as part of the development.

The only behavioral requirement is that FITH should not impact functionality of the tested driver. The tested driver should work as if there is no FITH at all.

There are various functionality requirements that need to be considered. Most center around the interception of resources access. FITH needs to have capability to intercept accesses for MMIO, IO, IRQ and PCI configuration. The other major requirement is about handling after interception. FITH needs to have capability to support the complex and customized post-handling, such as tracing hardware status, emulating fake hardware register, injecting error data and logging.

With respect to performance, there are basically two overriding goals:

- minimize the impact to system performance when the tested driver doesn't enable FITH.
- minimize the number of instructions in critical kernel path, such as exception and interrupt part.

3 Architecture

FITH consists of four components: interceptors, faultsets, configuration tools, and a fault injection manager. The configuration tools provide the command-line utilities needed to customize a faultset for a driver. Three interceptors will be implemented to catch IO, MMIO and IRQ access. When a driver tries to access a hardware resource, the IO interceptor captures this access and asks the fault injection manager if there is a corresponding item in the faultset. If there is, the fault injection manager determines how to inject the appropriate fault according to the associated properties defined in the faultset. The fault injection manager then returns this information to the interceptor, so the interceptor injects the actual fault into the hardware.

IRQ fault injection is somewhat different from the other types of fault injection. Hardware triggers the IRQ, and a kernel IRQ interrupt handler delivers this event to the IRQ interceptor. The IRQ interceptor then checks the faultset to determine whether a specific fault is available to inject into the event. 1 illustrates how the interceptor interacts with the other subsystems.

The interceptor sits between the hardware and the device driver and modifies information based on certain conditions in the driver's namespace. An interceptor for one driver does not affect the interceptor for others. As a matter of fact, the hardware that the driver observes is the hardware that our interceptor wraps.

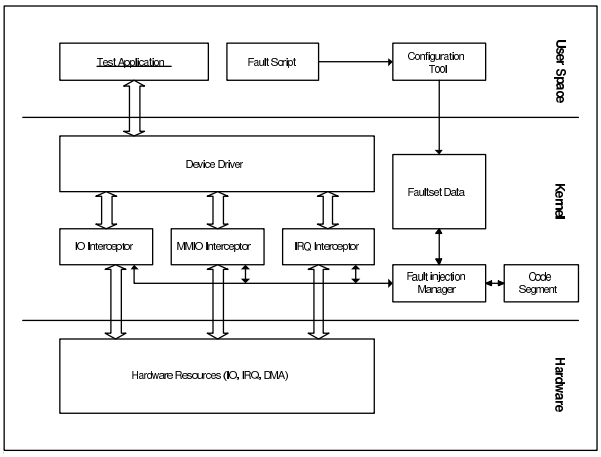

Figure 1: Architecture of FITH

4 KMMIO—Interceptor of MMIO access

One of those requirementsof FITH is the ability to hook to a specific memory mapped IO region before the user of the region gets access. A fault injection test case may need to just note when a given region is being read/written, take some action before the caller returns from the read or write operation, or change the value that is being read or written.

4.1 Approaches for capturing MMIO accesses

There are several hardware/software approaches for capaturing MMIO accesses.

- Overriding MMIO functions.

  Memory mapped IO access can be captured by overriding MMIO functions, such as `readb()` and `writeb()`. The major advantage is this method is platform-independent because all Linux platforms support these MMIO functions. Disadvantages are

  1. Any driver that accesses MMIO without using the standard MMIO functions cannot be intercepted.
  2. A special FITH header file needs to be added to the driver code, and the driver needs to be recompiled.
  3. There are some differences between the “released driver” and the “driver with FITH.” The driver that is validated and verified is the driver with FITH rather than the released driver.

- Setting Watch Points.

  IA-32 architecture provides extensive debugging facilities for debugging code and monitoring code execution. These facilities can also be used to intercept memory access. The major advantages are

  1. The driver does not need to be recompiled.
  2. There have been a good patch to support it.[3]

  On the other hand, there are several disadvantages:

  1. In IA32 architecture, this is a trap type of exception, which means that the processor generates the exception after the IO instruction has been executed, so this method cannot do fault injection in write operation.
  2. There are only four watch points that can be used. This may be not enough in a complex environment.

- Trapping MMIO access by using Page-Fault Exceptions.

  Like normal memory, MMIO is handled by a page-protection mechanism. Therefore, MMIO access can be intercepted by capturing page faults. The method clears the PE (PRESENT) bit of the PTE (Page Table Entry) of the MMIO address so that

the processor triggers a page-fault exception when MMIO is accessed. The major advantages are

1. In IA32 architecture, this is a fault type of exception, which means that the processor generates an exception before MMIO access is executed, so this method can do fault injection in write operation.
2. The driver does not need to be recompiled. There is a disadvantage in the method—because the unit of intercepted MMIO is the size of a page (4k in IA-32), an adjacent MMIO access may unnecessarily trigger an exception, system performance might be impacted.

Based on FITH requirements and our analysis above, we implemented a page-fault method to capture MMIO.

```
diff −Nru a/arch/i386/mm/fault.c
b/arch/i386/mm/fault.c
−−− a/arch/i386/mm/fault.c Thu May 15
15:52:08 2003
+++ b/arch/i386/mm/fault.c Thu May 15 15:52:08
2003
@@ −80,6 +81,9 @@
  /* get the address */
  __asm__("movl %%cr2,%0":"=r"
          (address));

+ if (is_kmmio_active() && kmmio_handler(regs,
address))
+     return;
+
  /* It's safe to allow irq's after cr2 has been saved */
  if (regs−>eflags & X86_EFLAGS_IF)
      local_irq_enable();
```

Figure 2: Patch against fault.c

4.2 Implementation

We also followed the same usage style like what kprobes provides, with a `register_kmmio_probe()` function for adding the probe, and a `unregister_kmmio_probe()` function for removing the probe. The `register_kmmio_probe` adds the probe into internal list and set the page which the probe is on as UNPRESENT. After `register_kmmio_probe`, any access on the page will trigger a page-fault exception and fall into KMMIO core.

To get the control in page-fault exception, we need some tweaks to `faults.c`. KMMIO needs to add an additional path here. When the page-fault falls into KMMIO, KMMIO looks up the fault address in probe hash list. If the fault address is one of probes, the `pre_handler` of the probe is called.

Then, KMMIO tries to recover normal execution. It sets the page as PRESENT. But KMMIO needs to do more than this. Because the probe should be re-enabled after current instruction which trigger the page-fault exception, KMMIO enables single-step before exiting page-fault exception. Similar to the change to `faults.c`, KMMIO needs to patch `traps.c` to get the control when the single-step exception is triggered.

After the instruction, which triggers the page-fault execption, is executed, a single-step execption is triggered and falls into KMMIO again. If there is a probe on the fault address, KMMIO calls the `post_handler` of the probe. Then KMMIO sets the page as UNPRESENT again to enable the probe on the page.

```
diff −Nru a/arch/i386/kernel/traps.c
b/arch/i386/kernel/traps.c
−−− a/arch/i386/kernel/traps.c Thu May 15
15:52:08 2003
+++ b/arch/i386/kernel/traps.c Thu May 15
15:52:08 2003
@@ −524,6 +525,9 @@

   __asm__ __volatile__("movl %%db6,%0" : "=r"
                        (condition));

+  if (post_kmmio_handler(condition, regs))
+      return;
+
   /* It's safe to allow irq's after DR6 has been saved */
   if (regs−>eflags & X86_EFLAGS_IF)
       local_irq_enable();
```

Figure 3: Patch against traps.c

5 KIRQ—Interceptor of IRQ handler

Placing hooks into IRQ handler of devices is a straight task. KIRQ stores IRQ handler of the device into `struct kirq` and modifies the IRQ chain in kernel to replace IRQ handler of the device with KIRQ's handler. When the device's interrupt falls into KIRQ, it calls `handler` of the hook. Based on return value of `handler` of hook, KIRQ calls the original handler of the device in turn.

6 Example

The serial device driver in Linux kernel was used in our fault injection trials. Four steps were involved in developing the fault injection tests:

1. Identify resources that needs to be fault injected.
2. Prepare the faultset data source.
3. Set up the test environment.
4. Run the workload and analyze the results.

6.1 Preparing the Faultset Data Source

FITH supports faultset scripts and action code segments. They can be used for customizing For example, a transmission error fault would modify data when the driver received the hardware status from the register. The faultset description looks like the following:

```
<?xml version="1.0" encoding="UTF-8" ?>
<fsml
 xmlns=
 "http://fault-injection.sourceforge.net/FSML/">
  <trigger id="2"
         type="r"
         len="1"
         addr="0x3FD"
         bitmask="0"
         min="0"
         max="0"
         skip="0"
         protection_mask="0"
         hz="0">
        <action code−segment="cs_001" />
  </trigger>
</fsml>
```

Figure 4: Faultset description example

The corresponding code segment looked like the following:

6.2 Setting Up the Test Environment

There are three steps:

1. load FITH kernel modules.

```
#include <fith/state_machine.h>

unsigned long pointer=0;
int inject_faults(struct
  context *cur) {
  //translate the bus address into linear address
  line_addr = fith_bus2line(pointer);

  //inject errors in data by going though
  //the device special
  //structure data
  // ...

  return 0;
};


/* cs_001 is called by trigger "001" in FSML
 * script when IO port 0x3FD
 * (Command Register) is written. */
int cs_001(struct state_machine *sm,
  struct context *cur) {
  unsigned long line_addr;
  if (cur->data==,a,) {
    // check if it is 'a' character
    inject_faults(cur);
  }
  return 0;
};
```

Figure 5: Code segment example

2. set up the faultsets by Fault Injection Command-Line (ficl) configuration tool.

3. load the serial driver.

6.3 Running the Workload and Analyzing the Results

To validate the serial driver, a well-chosen workload was run to stress the driver when it accessed the device. FITH injected faults between the driver and device and logged these operations. The fault-induced results and injected faults were later analyzed.

7 System Performance Impact

In this section we assess the performance impact of current implementation of FITH.

7.1 LMbench

LMbench is a general OS benchmark designed to measure all sides of OS from application view. This is useful for generating a set of apples to apples systems comparisons between pure Linux kernel and FITH-enabled Linux kernel.

All experiments were performed on a dual Pentium-III 933, 512K L2 Cache, 512 MB RAM system. The “52-pure” data was obtained by running on a vanilla 2.5.52 linux kernel. The “cs@1901” data was obtained by running on a patched 2.5.52 Linux kernel (which contained KMMIO KIRQ etc. patches). There are no active probes in “cs@1901” experiment.

Because FITH patches Linux kernel in page-fault exception path, the potential impacts should be in memory management subsystem. Current FITH implementation, however, has minimized the impact when there are no active probes. Differences between two experiments are so small that they are buried by testing noise.

8 Acknowledgements

We specially thanks following persons for their kind help and feedback:

David Edwards (for initial prototype and design), Frank Wang and Elton Yang (for project plan and support), Fleming Feng (for feedback

LMBENCH 2.0 SUMMARY

Basic system parameters			
Host	OS	Description	Mhz
52-pure	Linux 2.5.52	i686-pc-linux-gnu	932
cs@1901	Linux 2.5.52	i686-pc-linux-gnu	932

Processor, Processes - times in microseconds - smaller is better												
Host	OS	Mhz	null call	null I/O	stat	open clos	selct TCP	sig inst	sig hndl	fork proc	exec proc	sh proc
52-pure	Linux 2.5.52	932	0.39	0.68	22.9	24.4	27.2	1.09	4.47	210.	996.	5050
cs@1901	Linux 2.5.52	932	0.37	0.68	22.7	24.0	31.5	1.06	4.51	221.	1052.	5202

Local Communication latencies in microseconds - smaller is better									
Host	OS	2p/0K ctxsw	Pipe	AF UNIX	UDP	RPC/ UDP	TCP	RPC/ TCP	TCP conn
52-pure	Linux 2.5.52		7.109	33.4	41.7	61.6	81.1	110.2	110.
cs@1901	Linux 2.5.52		7.137	15.2	41.8	61.1	81.1	109.8	134.

File & VM system latencies in microseconds - smaller is better								
Host	OS	0K File		10K File		Mmap	Prot	Page
		Create	Delete	Create	Delete	Latency	Fault	Fault
52-pure	Linux 2.5.52	92.5	46.0	225.6	75.3	2053.0	0.885	2.00000
cs@1901	Linux 2.5.52	93.4	46.5	229.2	77.1	2063.0	0.765	2.00000

Local Communication bandwidths in MB/s - bigger is better										
Host	OS	Pipe	AF UNIX	TCP	File reread	Mmap reread	Bcopy (libc)	Bcopy (hand)	Mem read	Mem write
52-pure	Linux 2.5.52			40.7						
cs@1901	Linux 2.5.52			41.0						

Figure 6: LM Benchmark

and requirement), Rusty Lynch (for sysfs interface in FITH).

`http://www-124.ibm.com/linux/projects/kprobes/`

References

[1] David A. Edwards, “An Approach to Injecting Faults into Hardened Software,” Proceedings of the Ottawa Linux Symposium, `http://wwww.linuxsymposium.org/2002`

[2] Andy Chou, Junfeng Yang, Benjamin Chelf etc., “An Empirical Study of Operating System Errors,” 2001, `http://www.stanford.edu/~engler/metrics-sosp-01.ps`

[3] Vamsi Krishna, Rusty Russell etc., “Kernel Probes for Linux,”

Proceedings of the Linux Symposium

July 23th–26th, 2003
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
Stephanie Donovan, *Linux Symposium*
C. Craig Ross, *Linux Symposium*

Review Committee

Alan Cox, *Red Hat, Inc.*
Andi Kleen, *SuSE, GmbH*
Matthew Wilcox, *Hewlett-Packard*
Gerrit Huizenga, *IBM*
Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*
Martin K. Petersen, *Wild Open Source, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*