The Enhancement of Kernel Probing
- Kprobes Jump Optimization

Masami Hiramatsu
masami.hiramatsu.pt@hitachi.com
Hitachi Systems Development Laboratory
Linux Technology Center
Agenda

- Kprobes – Why it is useful
- Kprobes – How it works
- Performance Enhancing Ideas
  - Booster
  - Jump Optimization
- Technical Issues
  - Interrupts
  - Instruction Boundary
    - X86 Instruction Decoder
  - Jumps
  - Cross Code Modifying
- Implementation
  - Transparency of API/ABI
  - Greedy Optimization
  - Reserve Text
- Results
  - Kprobes
  - Kretprobes
  - Results on KVM
- Conclusion
Kprobes – What’s Kprobe?

• Kprobes is a dynamic software breakpoint function in the kernel
  – This allows you to add breakpoints inside kernel
    • User can check the kernel internal state almost anywhere
    • This allows user to tweak kernel internal state too (e.g. fault injection, and dynamic patching)
  – Dynamically add and remove the breakpoints.
  – Manage the breakpoint handlers
    • Handling breakpoint exception and call handlers
    • Aggregate probes on the same address
    • Disable probes when a target module is gone
    • Etc.
Kprobes - How it works

- Kprobes uses a breakpoint and a single-step

**Preparing**

Kernel code

1. Copy original and Modify rip-relative instruction
2. Put an int3

**Running**

Kernel code

1. Hit an int3
2. Invokes User pre_handler
3. Set TF=1
4. Trap single-stepping
5. Fixup registers and return to next instruction

User handler

Kprobes pre

Copy buffer

Kprobes post

Copyright (c) 2010 Hitachi LTD., Systems Development Lab. All right reserved
Motivation: Performance Issue

- Kprobes uses 2 exceptions
  - Software Breakpoint exception
  - Single-step trap

Normal kprobe consumes >1500 cycles/probe
Kprobe Booster

- Kprobes Booster skips Trap exception
  - Add a jump which jumps back to next instruction
  - Execute copied instruction and the jump
  - Some instructions can’t be boosted
    - Call, near jump, etc

Running

1. Hit an int3
2. Invokes User pre_handler
3. Return to the copied instruction
4. Jump Back to the next instruction
Jump Optimization

- Kprobes Jump Optimization
  - Skips software breakpoint too
    - No exception: Reduce the overhead drastically
    - It’s not easy – of course.
  - This will replace **several instructions** with one jump
    - Kprobes just replace one instruction.

### Running

1. **Hit a jump**
2. Store regs and call kprobe handler
3. Invokes User pre_handler
4. Restore regs and execute copied instruction
5. Jump Back to the next instruction

---

**Kernel code**

- Jump

**Optimized buffer**

- Reg save/call/restore
- Jump

**User handler**
• Interrupts can happen on other processors

Kernel code

insn1

insn2

jump

(1) Interrupt happens on the address of the middle instruction

(2) Another processor replaces the instruction with a jump

(3) Return to the middle of the jump instruction -> unexpected result!

Kernel code

jump

insn2

Bang

Make sure no process is interrupted on the address where will be replaced by the jump
• Make a bypass and wait for scheduler

(1) Interrupt happens on the address of the middle instruction

(2) Make a bypass code and put a breakpoint

(3) Wait for calling scheduler on all cpus

(4) Return from the interrupt handler

(5) Call scheduler at somewhere else (if NO preemptive kernel)

(6) Replace instructions with a jump
Boundary Issue

- x86 is a CISC processor
  - Instructions vary in length
  - How many bytes do we need to copy?

<table>
<thead>
<tr>
<th>Kernel code</th>
<th>insn1</th>
<th>insn2</th>
</tr>
</thead>
</table>

How many bytes?

- Check non-relocatable instructions
  - Some IP-related instructions can’t execute directly on copy buffer (Call, relative-jump, etc)
  - How can we find those instructions if it is in the middle?

<table>
<thead>
<tr>
<th>Kernel code</th>
<th>insn1</th>
<th>IP-related</th>
</tr>
</thead>
</table>

From where does it start?

We need something to decode instructions!
x86 Instruction Decoder

- Introduce in-kernel x86 instruction decoder
  - Simple instruction decoder
    - Just ~350 logical lines including AVX (Intel® Advanced Vector Extensions) decoding support
  - Generic & easy maintain
    - Based on x86 opcode map (in Intel’s software developers manual)
    - Generate instruction attribute map from the opcode map when compiling kernel

![Diagram of build time generation and kernel binary]

Copyright (c) 2010 Hitachi LTD., Systems Development Lab. All right reserved
• x86 instruction decoder has two parts
  – insn
    • Data structure represents an instruction
    • insn_init() and insn_get_XXX()
    • users usually use this part
  – inat
    • Instruction attribute maps for decoding
      – Each opcode has attributes

```c
struct insn;
int x86_64 = 0; /* depends on the arch */
insn_init(&insn, target_address, x86_64);
insn_get_length(&insn); /* insn_get_length() decodes the entire instruction */
printk("opcode size:%d, instruction length:%d\n", insn.opcode.size ,insn.length);
```
There are some jump-in issues

- Kernel jumps into the middle of target instructions

- Kernel MAY jump into the middle of target instructions
Jump Code Analysis

- Check a target function to find those jumps
  - Decode an **entire function**
  - Check pagefault fixup table

Kernel code (a function)

Function Entry

(1) step-by-step decoding

(2) jump destination analysis

(3) indirect jump searching

(4) Fixup table searching

Function End

- Reject optimization and just use normal kprobe
  - If a jump destination is the middle of target
  - If the function including indirect jump
  - If the function including an address in fixup-table
Cross Modifying Code (Self Modifying on MP)

• Cross modifying code needs a special operation
  – Documented method
    • Intel® 64 and IA-32 Architectures Software Developer’s Manual Vol. 3 8.1.3
  – Stop-machine and modify code
    • This can’t use in NMI handler, but kprobes itself doesn’t allow to probe NMI handler too.
  – Stop-machine is slow, so modifying should be batched.

(1) Stop other processors

```
 CPU0  --- stop --- CPU1  --- ... --- CPUN
```

(2) Write a jump

```
 insn1   insn2

datastructure
```

(3) Serializing and continue to run on other processors
• Int3 bypass method
  – Make a bypass by using int3 while XMC
  – No stop machine required
  – Still be under discussion

(1) Make a bypass

(2) Write a jump destination and sync all processors (send IPI)

(3) Write a jump opcode and sync all processors (send IPI)
Transparent API

- **Optimization without changing APIs**
  - Optimized kprobe is hidden in `aggr_probe`
    - `Aggr_probe` is usually used for aggregating multiple probes on the same address
  - User don’t know their probe is optimized or not.

![Diagram showing Normal Aggregation and Optprobe processes](image-url)
**Transparent Optimization**

- Optimization is transparently done (No explicit APIs)
  - Jump code modifying is done in background
  - Some probe state changes requires unoptimizing
    - Unoptimizing is also done in background
  - Only one knob for debugging
    - `/proc/sys/debug/kprobes-optimization`
• Optimizing/Unoptimizing probes automatically
  – Kprobes tries to optimize probes every state change if possible
    • A probe removed from the instruction next to another probe
    • An aggregated probe which has a post_handler is removed

Kernel code

```
int3 jump
```

```
new probe
```

```
unoptimize
```

```
optimize if possible
```

```
new optprobe
```

```
remove
```

```
optimize again
```

```
```

```
```

Copyright (c) 2010 Hitachi LTD., Systems Development Lab. All right reserved
Some other functions can modify text too
  - Ftrace, alternatives, jump labels
  - Only kprobes is modifying code anywhere
  - Introduce text_reserve interface
    - Checking specified area can be modified by other functions
    - If so, kprobes gives up putting a probe on it.
Results

- Performance results (unit is cycles)

![Bar Chart]

- Optimization can reduce the overhead to ~100 cycles
- Kretprobe is also optimized
Results on KVM

- Performance results on KVM
  - On KVM, kprobes is much heavier, because trap is emulated

Optimized and boosted probes can run inside guest.
What is the benefit of less overhead?

• Lower overhead allows us to trace more events
  – Tracing overhead breakdown
    • Probing overhead (depends on optimization)
    • Recording overhead (~300 cycles)
  – Total ~400 cycles overhead/event allows us to trace 100K events/sec with just 1~2% overhead on 3GHz CPU
Conclusion

- **Kprobes**
  - Dynamic/Flexible in-kernel probing function
  - But heavy, especially with Virtualization

- **Kprobe jump optimization**
  - Drastically reduce overhead of kprobes
  - Some limitations
  - Transparent optimization
    - User need nothing to change
  - Good performance with Virtualization
History of Kprobes Jump Optimization

- Long history of kprobes jump optimization
- 2005 May: Got an idea for jump optimization
- 2005 Jul: First Prototype Release
- 2005 Aug: 1st Upstream Try
- 2006 Oct: 2nd Upstream Try
- 2007 Jul: 1st Presentation of “djprobe” in OLS
- 2008 – silent but things going forward…
- 2009 Jun: x86 instruction decoder Release
- 2009 Jun: Revised “Optprobe” Release
- 2010 Feb: Optprobe is merged!
Related Articles

- Minimizing instrumentation impacts (kprobes jump optimization)
  - http://lwn.net/Articles/365833/
- Kernel documents
  - Documents/kprobes.txt
Thank you!
Trademarks

- Linux is a trademark of Linus Torvalds in the United States, other countries, or both.
- Intel and Core are trademarks or registered trademarks of Intel Corporation in the United States, other countries, or both.
- Other company, product, or service names may be trademarks or service marks of others.