# Kernel Probes for ARM

Quentin Barnes
q.barnes@motorola.com
Motorola - Mobile Devices
April 17, 2007

# Overview

- Introduction to kernel probes
- How kernel probes work
- The classic kprobe model and 'boosting'
- Difficulties with the classic model and ARM
- New approach used for ARM
  - Details of implementation
  - Cost and Performance
- Wrap-up

# Kernel Probes Introduction

- New with 2.6.9-rc2 kernels
  - Currently: avr32, i386, x86_64, ia64, powerpc, sparc64, s390
- Allows dynamic breakpoints (registered at runtime) to be placed most anywhere in kernel
- Breakpoint transfers control to user specified handlers, executes the instruction, and returns
- User handlers often used to aid debugging or collect performance data
- Can be relatively low overhead

# Types of Kernel Probes

- Kprobes - “Kernel Probes”
  - Places breakpoint at symbol plus offset or address
- Jprobes - “Jumper Probes”
  - Insert function call ahead of probed function
- Kretprobes - “Return Probes”
  - Probe function's return
- Djprobes - “Dynamic Jump Probes”
  - Future direction: Kprobe-like, uses a jump instruction rather than a breakpoint

# How Do Kprobes Work?

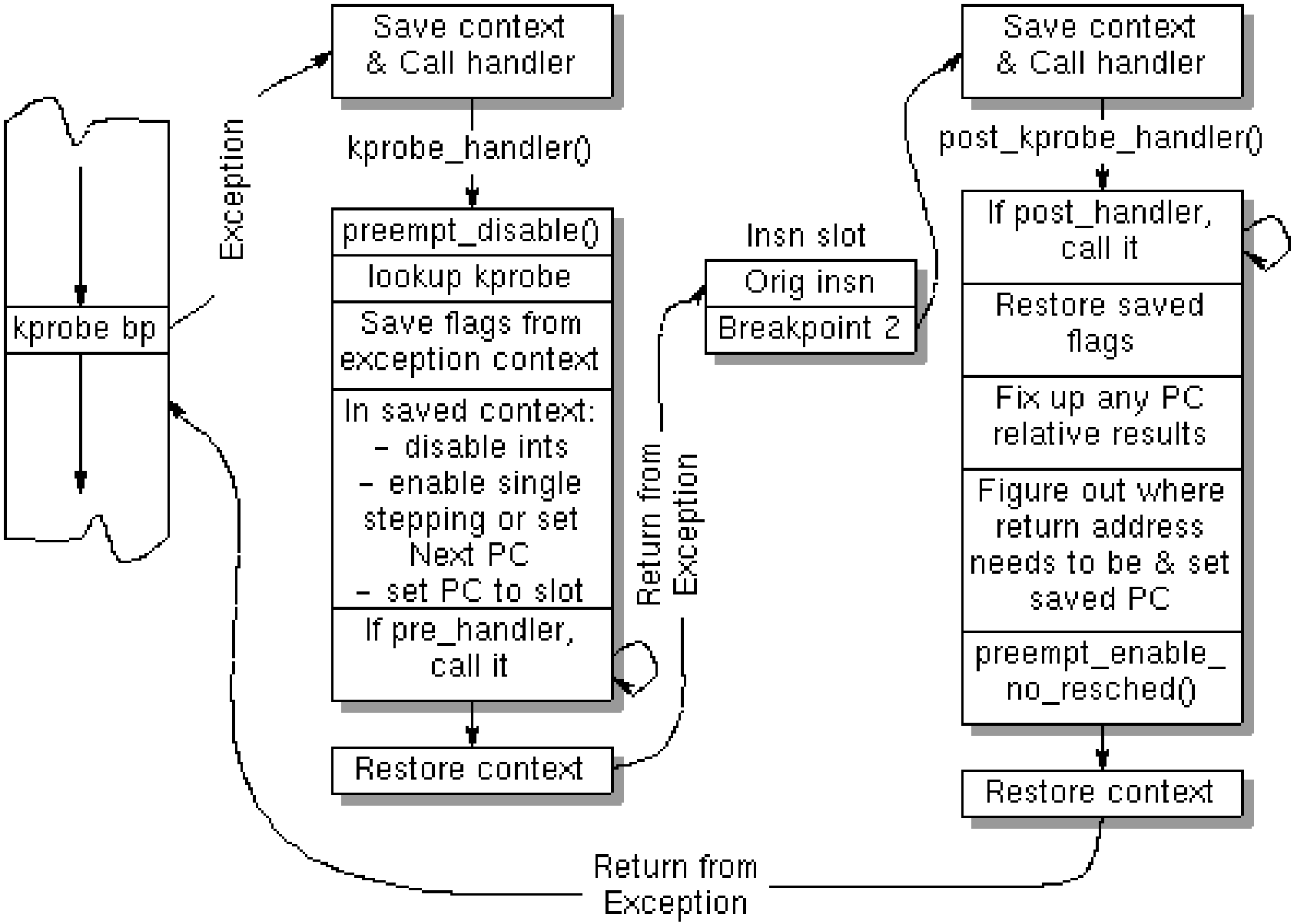
- Places a breakpoint instruction most anywhere
  - On ARM, uses an undefined instruction
- May register optional user pre-, post-, break, and fault handlers
  - User handlers can do most things except block
- More than one Kprobe allowed at one address
  - Not so for Jprobes and Kretprobes

# 'Classic' Kprobes Model

# Kprobes Registration & Prep

- Do symbol lookup, if necessary
- Validate address
- Record probe and address for hash lookup
- Save instruction at address and replace it with a breakpoint instruction
- Allocate and initialize instruction slot
- Do additional architecture specific initialization

# Executing a Kprobe

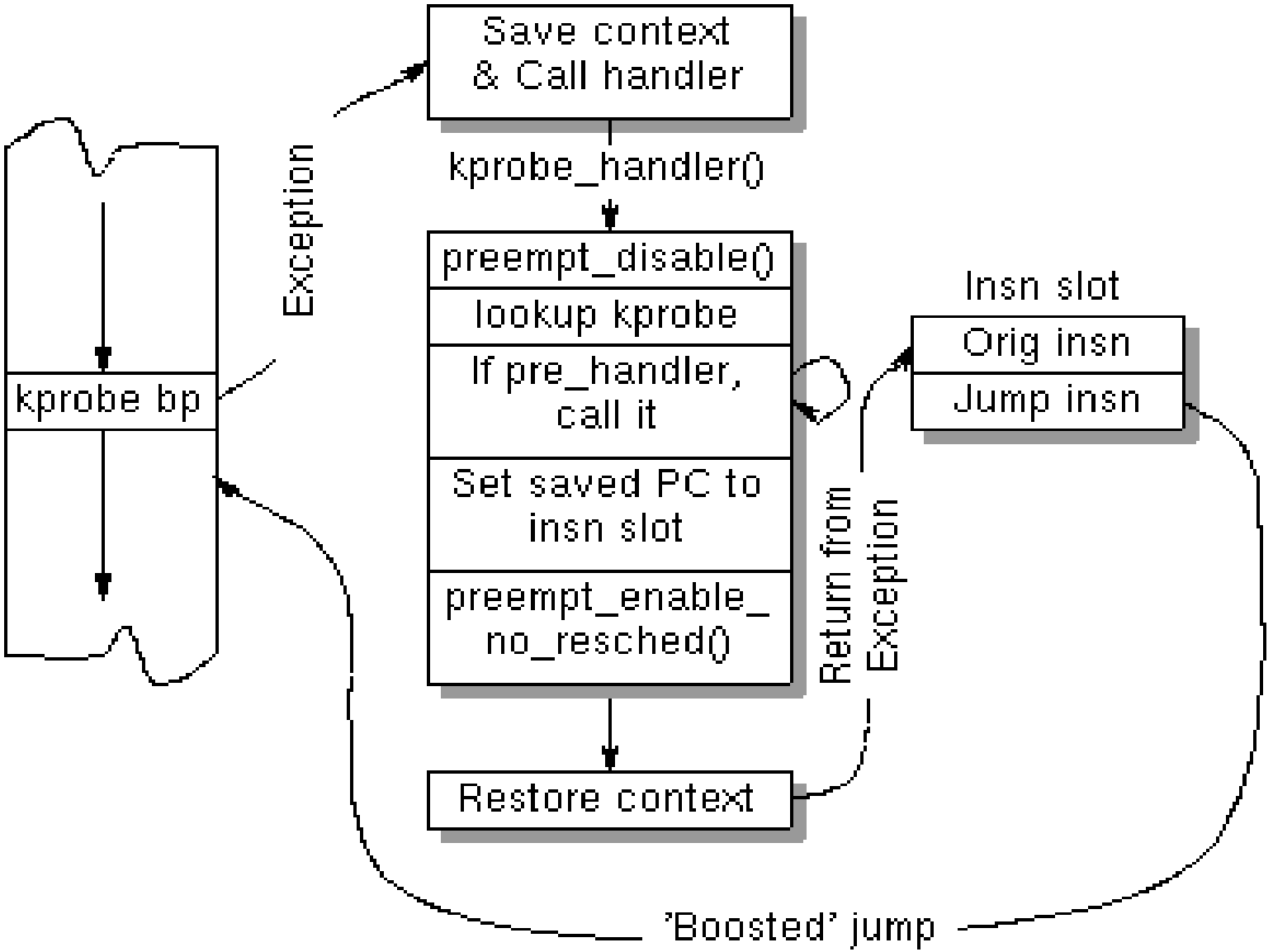

Save context
& Call handler
kprobe_handler()
preempt_disable()
lookup kprobe
Save flags from
exception context
In saved context:
– disable ints
– enable single
stepping or set
Next PC
– set PC to slot
If pre_handler,
call it
Restore context
Exception
kprobe bp
Insn slot
Orig insn
Breakpoint 2
Return from
Exception
Save context
& Call handler
post_kprobe_handler()
If post_handler,
call it
Restore saved
flags
Fix up any PC
relative results
Figure out where
return address
needs to be & set
saved PC
preempt_enable_
no_resched()
Restore context
Return from
Exception

# Issues with Classic Model

- High execution overhead - two exceptions and four full context switches
- Holds interrupts disabled through entire length
  - Some re-enable though resulting in possible unbounded recursion, unpredictable behavior, and skipped user handlers

# Performance Gain – 'Boosting'

- Take one exception instead of two
- Boosting short-circuits the second exception for pre-determined instructions by jumping directly back after execution


Save context
& Call handler
kprobe_handler()
preempt_disable()
lookup kprobe
If pre_handler,
call it
Set saved PC to
insn slot
preempt_enable_
no_resched()
Restore context
Exception
kprobe bp
Insn slot
Orig insn
Jump insn
Return from
Exception
'Boosted' jump

# Boosting Limitations

- Limited to only 'boostable' instructions
  - No conditional jumps and no reserved or special instructions
- Non-boostable instructions still run classic way (using two exceptions)
- Use of post-handlers disable boosting still running classic way
- Garbage collector

# Boosting's Garbage Collector

- Boosting holds lock on instruction slot even after execution completes
- Garbage collector needed to clean up unused locks
- When kernel preemption is enabled, garbage collector disables all interrupts and freezes all kernel threads across all CPUs simultaneously to run its O(n) time algorithm

# Difficulties with Kprobes & ARM

- No single-stepping supported in the processor
  - Single-stepping causes the processor to trigger an exception after completing the current instruction
- No Next-PC register
  - The Next-PC register steps the processor to the “next” address after completing the current instruction

# ARM Difficulties (cont.)

- So no way to regain control of the ARM processor after executing instructions that alter the PC without decoding, detecting, and handling all instructions that can write the PC
- No way to give correct results for instructions that read the PC without decoding, detecting, and handling
- Once you've gone that far already, it's not that much further to a new approach

# New Kprobes Approach

# New Kprobes Goals

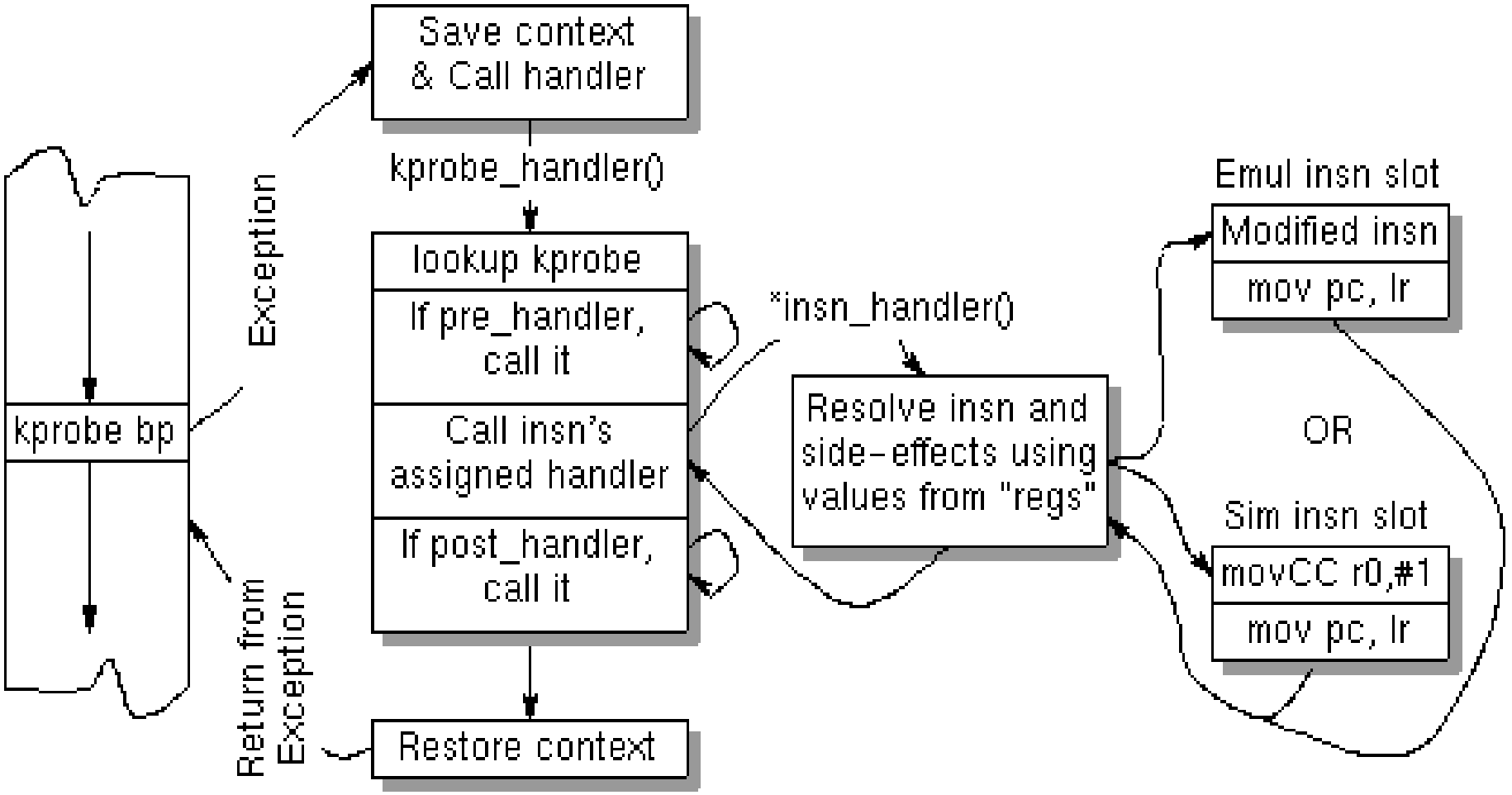
- Do as much prep work as possible during registration of a kprobe to reduce its execution overhead
- Complete the kprobe'd instruction and all its side-effects before returning from first (and only) exception
- Provide consistent execution and behavior
- Don't hold any locks longer than necessary

# New Kprobes Advantages

- Fully self-contained within arch, no changes to generic Kprobes code needed
- Can work for all instructions, not just some
- Always works, regardless of post-handlers
- No second exception ever, so all that support code and additional logic goes away
- No locks left dangling – so no headaches and no garbage collector

# Advantages (cont.)

- Interrupts disabled throughout single exception
  - No unbounded recursion
  - User handlers guaranteed to be called unless a handler itself triggers recursion by having a kprobe
- Very light-weight
  - Small code size
  - Low execution overhead
- MP and Preemptive (CONFIG_PREEMPT) clean


Save context
& Call handler
kprobe_handler()
lookup kprobe
If pre_handler,
call it
Call insn's
assigned handler
If post_handler,
call it
Restore context
Exception
Return from
Exception
kprobe bp
*insn_handler()
Resolve insn and
side-effects using
values from "regs"
Emul insn slot
Modified insn
mov pc, lr
OR
Sim insn slot
movCC r0,#1
mov pc, lr

# How Does *insn_handler() Work?

- At registration time:
  - Depending on the form of the kprobe'd instruction, an instruction execution handler is assigned
  - Registers used by the instruction are recoded to use the registers the instruction handler expects
  - Modified instruction or condition code test instruction saved to instruction slot
- Two groups of instruction handlers - ones that execute a modified form and ones that resolve the instruction in straight C code
  - C-only ones are for instructions that write the PC

# Instruction Handler: Example 1

- Kprobe'd instruction - “blne *addr*”
- Modified instruction in slot - “movne r0, #1”
- Handler assigned - simulate_bbl()
  - 25 total instructions

# simulate_bbl()

```
static void __kprobes simulate_bbl(struct kprobe *p, struct pt_regs *regs)
{
        insn_1arg_fn_t *i_fn = (insn_1arg_fn_t *)&p->ainsn.insn[0];
        kprobe_opcode_t insn = p->opcode;
        long    iaddr = (long)p->addr;
        int     disp  = branch_displacement(insn);

        if (!insnslot_1arg_rflags(0, regs->ARM_cpsr, i_fn)) return;

        if (insn & (1 << 24)) regs->ARM_lr = iaddr + 4;

        regs->ARM_pc = iaddr + 8 + disp;
}
```

- All in C since “bl” writes to the PC
- Handles both “b” (branch) & “bl” (branch & link)
- Could have been two handlers

# insnslot_1arg_rflags()

```
static inline long __kprobes insnslot_1arg_rflags(long r0, long cpsr,
                                        insn_1arg_fn_t *fn)
{
        register long rr0 asm("r0") = r0;
        register long ret asm("r0");

        __asm__ volatile(
                "msr    cpsr_fs, %[cpsr]        \n\t"
                "mov    lr, pc                  \n\t"
                "mov    pc, %[fn]               \n\t"
                : "=r" (ret)
                : "0" (rr0), [cpsr] "r" (cpsr), [fn] "r" (fn)
                : "lr", "cc"
        );
        return ret;
}
```

# insnslot_1arg_rwflags()

```
static inline long __kprobes insnslot_1arg_rwflags(long r0, long *cpsr,
                                                   insn_1arg_fn_t *fn)
{
        register long rr0 asm("r0") = r0;
        register long ret asm("r0");
        long    oldcpsr = *cpsr;
        long    newcpsr;

        __asm__ volatile(
                "msr    cpsr_fs, %[oldcpsr]     \n\t"
                "mov    lr, pc                  \n\t"
                "mov    pc, %[fn]               \n\t"
                "mrs    %[newcpsr], cpsr        \n\t"
                : "=r" (ret), [newcpsr] "=r" (newcpsr)
                : "0" (rr0), [oldcpsr] "r" (oldcpsr), [fn] "r" (fn)
                : "lr", "cc"
        );
        *cpsr = (oldcpsr & ~PSR_fs) | (newcpsr & PSR_fs);
        return ret;
}
```

# Instruction Handler: Example 2

- Kprobe'd instruction - “add r3,r6, #7”
- Modified instruction  - “add r0, r0, #7”
- Handler assigned – emulate_alu_imm_rflags()
  - 19 total instructions

# emulate_alu_imm_rflags()

```
static void __kprobes emulate_alu_imm_rflags(struct kprobe *p,
                                             struct pt_regs *regs)
{
        insn_1arg_fn_t *i_fn = (insn_1arg_fn_t *)&p->ainsn.insn[0];
        kprobe_opcode_t insn = p->opcode;
        int     rd  = (insn >> 12) & 0xf;
        int     rn  = (insn >> 16) & 0xf;
        long    rnv = (rn == 15) ? (long)p->addr + 8 : regs->uregs[rn];

        regs->uregs[rd] = insnslot_1arg_rflags(rnv, regs->ARM_cpsr, i_fn);
}
```

# Isn't It Expensive?

- Twenty-seven instruction handlers (5 sim + 22 emul) cover virtually all ARM instructions
- Most instruction handlers compile down to just 10-40 instructions
- The ARM Kprobes is 6.6KB of code
- With no second exception ever and no boosting, all that support code is tossed away

# Overhead Performance

- Overhead results on PXA270@364.00MHz
  - Kprobe: 1.21us
  - Jprobe: 3.11us
  - Kretprobe: 3.40us
  - Kprobe + Kretprobe: 3.81us
  - Jprobe + Kretprobe: 5.86us

- Measured with Jim Keniston's kprobe overhead testing software with defaults (iter=5000000, handler=1)

# So What's the Catch?

- Uses a lot of decode decision paths and some handlers infrequently used, so needs extensive testing and chance of latent bugs
- Technique is easiest for RISC and orthogonal instruction sets, but could be harder for others

# State of the Port

- Kprobes, Kretprobes, and Jprobes complete and all working
- Some bugs still running around
- Some rare and unusual instructions not supported – ones that change CPSR state
- Patches for 2.6.20.4 kernel released to Systemtap/Kprobes mailing list on 4/11/07

# What's Left

- Shakeout of approach and code
- Test suite to test all decode paths and instruction execution handlers
- Integrate into ARM Linux and kernel.org trees
- Finish port of Systemtap's runtime and test suite to ARM and modify Systemtap to support cross-compiles
- Djprobe version

# What's Left? (cont.)

- Remember for the new Kprobes I said one of the goals was to do as much prep work as possible?  I'm not quite doing that yet...
- Use larger instruction slots, do more at registration and less at runtime:

```
Original Inst       add    r3, r6, #7

Instruction         ldr    r1, [r0, #24]
slot                add    r1, r1, #7
                    str    r1, [r0, #12]
                    mov    pc, lr
```

# Systemtap

- User space front end to Kprobes
- Simple command line interface and scripting language
- Does it all – compiles script to C code, builds, loads, and runs the kernel module

# URLs

- Systemtap:
  - Home: http://sourceware.org/systemtap/
  - Mailing list: http://sourceware.org/ml/systemtap/
- Motorola Open Source
  - http://opensource.motorola.com/

Questions?