Getting started with Virtual Platforms:
A Software Developer Perspective
Virtual Platform Workshop
DAC 09, San Francisco
Objective: Provide illustrative examples on how Virtual Platforms are used for debugging.

Non-Objective: Provide a complete feature- and benefit-list of Virtual Platforms.
Outline

- SW Developers‘s Views
- Debugging on the boundary of HW and SW
- Virtual Platform based Debugging
- Platform Level Software Analysis
- Virtual Platform Scripting
  - Tracing
  - Software Assertions
- Summary
Embedded Software Design Tasks

- Porting/Bring-Up/Development
- Pre-/Post-Silicon Validation
- Optimization
- Debugging

Presentation focus
User's View

[DEMO: Android Phone UI with Android AMP skin]
Example: Device Key Press
Example: Device Key Press

What if something goes wrong?
Debugging Process

Start

Run & validate software

Prepare debug iteration (e.g. earlier breakpoints)

Expected Behavior?

Analyze symptom
What worked?
What did not work?

Analyze cause
Why did it not work?

Adapt code
Modify & build software

Potential cause identified?

Done

Yes

Yes

No

No

Yes

No

Start

Prepare debug iteration (e.g. earlier breakpoints)

Adapt code
Modify & build software

Potential cause identified?

Done

Yes

Yes

No

No
Time and Space During Debug

Software initiated system activity:
- Example: Timer dies
- What code should I look at and debug?

Hardware initiated system activity:
- Example: Device key press
- What time should I stop to debug?
Watchpoints: Fast Track to the Problem

- **Getting to the code:**
  - Need to determine the code that configures/corrupts the peripheral
  - A: Software access watchpoint

- **Getting to a point in time**
  - Need to closely investigate the SW reaction on an event generated by the HW
  - B: Hardware access watchpoint
  - C: Signal watchpoint
Watchpoints
Platform Level Software Debugging

E.g.:
ARM926EJS_0 step_core
VIC_1/EnableMask get_value

Virtual Platform Analyzer

TCL Scripting Interface

Control & Inspection Layer

Core #1 simulator

VIC 1

GPIO 1

VIC 1

VIC

VIC 2

GPIO 2

Core #2 simulator

Timer

E.g.: ARM926EJS_0 step_core VIC_1/EnableMask get_value

Break-/Watch-point

Visible Code

Visible Register

Visible Signal

SW Debugger

Mapped Registers

VIC

Internal Registers

Core

Core Registers

Bus Port

Bus Port

CoWare®
Platform Level Software Analysis

- **Debugging**: Analyze a snapshot of the system state
- **Challenge**: Understand/analyze system history
  - Interaction between HW and SW entities over time
- **Requirement**: System level tracing of HW/SW

- Domain A
  - Applications
  - Middleware
  - Drivers
  - OS/Bootloader

- Domain B
  - Applications
  - Middleware
  - Drivers
  - OS/Bootloader

- Domain C
  - Firmware

(Multi-Core) Hardware Platform
Platform Level Software Analysis

OS Context Tracing Core A
- decompress
- boot
- irq_timer
- kthread

OS Context Tracing Core B
- boot
- idle
- soft_irq
- h264_decode

Context’s Function Tracing
- asm_do_IRQ
- irq_enter
- handle_level_irq
- vic_mask_irq

Instruction Tracing
- c002b62c: MOV R0, R4
- c002b630: MOV R1, R5
- c002b634: MOV R2, R6
- c002b638: MOV R14, R15
- c002b6ec: LDR R15, [R5, #0]

Control & Inspection Interface
- Core #1 simulator
- Core #2 simulator
- TIMER 1
- VIC 1
- MEM 1
- TIMER 2
- VIC 2
- MEM 2

Platform Memory Access Tracing
- TIMER 1
- VIC 1
- MEM 1
- Shared MEM

Shared Memory
Virtual Platform Scripting

Scripting Use Cases:

- Deterministic repetition of scenarios
- Regressionizing
- Analyzing
- Debugging

E.g.:
ARM926EJS_0 step_core
VIC_1/EnableMask get_value

Virtual Platform Analyzer

TCL Scripting Interface

Control & Inspection Layer

VIC 1
Core #1 simulator
GPIO 1
Timer

VIC 2
Core #2 simulator
GPIO 2
Timer
Scripting For Debug

Principle:
- Notify and react on system events
  - Register, memory, pin access and change
  - Program control (e.g. Function call)
- Inspect state
  - Register, memory and pin values
- Validate
  - Assert correctness
- Feedback assertion result
  - Stop or carry state to next assertion
Typical Linux Boot Problem

1) Linux Boot UART Console

- Symptom: It does not boot!
- What worked, what did not work?
- No helpful kernel debug messages?
- UART driver not yet working!
- Debug messages would be helpful!

2) VP Debug Helper Script (TCL)

# 1: Create breakpoint at debug emit function
set bp [create_breakpoint "emit_log_char"]

# 2: Attach callback procedure
$bp set_callback tcl_emit_log_char

# 3: Callback procedure: print character from CPU register
proc tcl_emit_log_char {} {
    set c {i_ARM926EJS_0/R/R[0]} get_value
    puts -nonewline "[format "%c" $c]"
}

3) Kernel Debug Messages via VP

Virtual Platform Simulation Control & Inspection Interface

Efficient debugging via non-intrusive control, visibility and scripted automation.
Increased debug productivity!
**Scripting For Debug**

**Principle:**
- Notify and react on system events
  - Register, memory, pin access and change
  - Program control (e.g. Function call)
- Inspect state
  - Register, memory and pin values
- Validate
  - Assert correctness
- Feedback assertion result
  - Stop or carry state to next assertion
“My kernel shows sporadic kernel panic problems. How can I assert a memory corruption in the kernel?”

Memory corruption defects severely increase the system vulnerability.

### Kernel Memory Corruption Categories

<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Faulty heap memory mgmt.</td>
<td>Read un-initialized memory</td>
</tr>
<tr>
<td></td>
<td>Buffer overrun</td>
</tr>
</tbody>
</table>

### Kernel Memory Corruption Symptoms

<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory leaks</td>
<td>Kernel heap corruption</td>
</tr>
<tr>
<td></td>
<td>Most times immediate unpredictable kernel behavior</td>
</tr>
<tr>
<td></td>
<td>Delayed crash of kernel activities (scheduler, threads...)</td>
</tr>
</tbody>
</table>

#### History of a buffer overrun defect

```
char * tmp = (char*) malloc(strlen(str));
strcpy(tmp, str);
```

```
kthread
khelper
irq
```

#### Typical code prone to a buffer overrun

```
```
Kernel Memory Corruption

Linux kernel function:
`slob_page_alloc(int size, int align, int addr)`

- Probe parameters: address, size
- Set watchpoint at address + size

Virtual Platform Callback (Pseudo Code)

```c
set watchpoint
```
Kernel Memory Corruption

Demo: Linux SLOB (Simple List Of Blocks) Allocator – Virtual Platform Assertion
**Multi-core Driver Example**

**H.264 Stream Driver Example**

- **cat video.264 > /dev/h264**

**H.264 Stream Driver**

- **OS A**
- **CPU 0**
- **IRQ/Timer**
- **Memory**

**H.264 Decoder Firmware**

- **CPU 2**
- **IRQ/Timer**
- **Memory**

**Shared Address Space**

- **driver**
- **firmware**

- **start offset**
- **size**
- **write pos**
- **read pos**

**H.264 Stream Device Example**

- **Linux H.264 stream device**
  - Driver on **CPU 0**, provides data to
  - **H.264 decoder firmware on CPU 2**

**Control/Synchronization**

- **Interrupts**
- **Mutex/Semaphores**

**Data streaming**

- **Circular buffer in shared memory**
Multi-core Driver Example

Shared Address Space – Circular Buffer

<table>
<thead>
<tr>
<th>start offset</th>
<th>size</th>
<th>write pos</th>
<th>read pos</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

• Buffer overrun
• Race condition/data corruption
• Starvation

Debugging challenges
• Become aware of a defect
• Sporadic decoding errors (e.g. frames dropped)

System level software assertion
• Protected address regions,
• through VP region watch-points
• Dynamically adjusted,
• on every circular buffer update
Multi-core Driver Example

Shared Address Space

<table>
<thead>
<tr>
<th>start offset</th>
<th>size</th>
<th>write pos</th>
<th>read pos</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
<td>empty</td>
<td>(to be read)</td>
<td>(to be written)</td>
</tr>
</tbody>
</table>

Virtual Platform Callback Pseudo Code

- `update_watch_region_cpu_0`
- `update_watch_region_cpu_1`
- `CPU_0 set_range_watchpoint read $start_offset $write_pos-1`
- `CPU_1 set_range_watchpoint write $write_pos $read_pos-1`

System level software assertion

- Watch-points on circular buffer pointers
- trigger updates of the watch-regions.
- Platform execution is suspended,
- as soon as memory is accessed illegally.
Multi-core Driver Example
Multi-core Driver Example

```c
break;

if (j < i)
    continue;

for (q = 0; q < 2; q++) {
    int shift = div6[q] + 2;
    int idx = rem6[q];
    for (x = 0; x < 16; x++)
        h->dequant4_coeff[i][q][transpose ? (x >> 2) : ((x << 2) & 0xf) : x] =
            ((uint32_t) dequant4_coeff_init[idx][(x & 1) + ((x << 2) & 1)] *
            h->pps.scaling_matrix[i][x]) << shift;
}
```

Program stopped at line 3300

```c
next_write = (next_write + count) & mask;

amp_comm_control.buffer.next_write = next_write;

amp_write(AMP_BUFFER_NEXT_WRITE, next_write);

if (next_write == ((next_read - 1) & mask)) {
    printk("amp_comm: Signaling decoder\n");
    /* FORCE DECODER RUNNING */
    amp_write(AMP_COMMAND, AMP_CMD_START_CONTINUE);
}
```

Program stopped at line 390
Summary

We have used Virtual Platforms...
- to identify, analyze and assert software defects.
- by means of real-world hardware and software examples.

We have seen Virtual Platforms...
- provide non-intrusive and deterministic
  • Control & Visibility
- enable novel debug solutions,
- with less guessing and more analysis,
- resulting in increased productivity,
- for embedded software development.
Thank You!