A Run-Time Memory Protection Methodology

Udaya Seshua, Nagaraju Bussa*, Bart Vermeulen
NXP Semiconductors, *Philips Research

12th Asian and South Pacific Design Automation Conference 2007
January 25, 2007, Yokohama, Japan
Agenda

• Introduction
• Motivation
• Debugging Run-Time Memory Corruption
• Prior Work
• Proposed Debug Methodology
  – Hardware Design
  – Software Design
• Experimental Results
• Conclusion
Introduction

- System chips are becoming more and more complex
  - More transistors per mm\(^2\), customer requirements, embedded processors & SW, mixed processes...

- Code Size Evolution of High End TV Software

- # Transistors per die

- Source: Intel, ITRS roadmap

- Source: Rob van Ommering, PRLE Informatica Colloquium, October 2005
Introduction

- Extensive pre-silicon verification
  - Formal Verification
  - Simulation
  - Timing Verification
  - Emulation
  - DRC, LVS …

- No guarantee that all HW and SW errors are removed before silicon
  - Too many use cases
  - Mandatory trade-off between amount of detail and speed

- Debugging embedded software on prototype silicon is a necessity
  - Find remaining SW and HW errors

Effort as % of Project Time

Design 53%
Verification 47%

source: Collet International Research Inc.

Industry Silicon Spins

source: Numetrics Management Systems, Inc.
Motivation

• In any application nearly 70% of code deals with memory transfers
• Memory-related bugs are among the most prevalent and difficult to catch
  – particularly in applications written in an unsafe language such as C/C++
• In an embedded system, a single memory access error can cause an application to behave unpredictably or even a delayed crash
• A good debug infrastructure capable of locating memory-related bugs quickly is key to reducing the effort spent on software debug
Debugging Run-Time Memory Corruption

• A single incorrect memory access can crash an application and/or threaten its security

1. Fetch Pointer Value

2. Access data referenced by pointer

Debugging Run-Time Memory Corruption

- A single incorrect memory access can crash an application and/or threaten its security.

1. Fetch Pointer Value
2. Access unintended data referenced by *corrupted* pointer

How do we detect these errors efficiently at run-time?
Prior Work

• Mostly software-only methods ("Purify, xGCC and the like")
  – High performance penalty (5-10x not uncommon)
  – Not acceptable in real-time, embedded systems

• Available HW support often used on ad-hoc basis
  – a Memory Management Unit
  – a Processor data breakpoint

• “Whatever is available can and will be used!”
  – Even if it wasn’t designed for this purpose

• Results in long and unpredictable debug times
  – Slipping deadlines, market and possibly customer loss
Proposed Debug Methodology

• Structured Integrated Hardware/Software Approach
  – Monitor memory accesses of an application
    • Flag invalid accesses for QoS, security or debug
  – Perform frequently recurring tasks in hardware
    • Compare memory addresses with valid regions
  – Keep configurability in software for flexibility
    • Configure valid regions

• Make optimal trade-off between
  – Hardware cost, i.e. silicon area
  – Software cost, i.e. performance drop
Proposed Debug Methodology

Run-Time Memory Protection Architecture

```c
main()
{
    func1();
    func2();
}

func1()
{
    p = malloc(127);
    int a[10], b[10];
    free(p);
    a[10] = 0;
}

func2()
{
    int a[10], b[10];
}

rpu_id = 1;
main()
{
    rpus_initialize(0);
    func1();
    func2();
    rpus_check_access(a+10);
    a[10] = 0;
    rpus_stack_disable(id);
}

func1()
{
    id = rpu_id++;
    p = malloc(127);
    rpus_heap_enable(127, p);
    int a[10];
    rpus_stack_enable(10, a, id);
    int b[10];
    rpus_stack_enable(10, b, id);
    free(p);
    rpus_heap_disable(p);
    rpus_check_access(a + 10);
    a[10] = 0;
    rpus_stack_disable(id);
}

func2()
{
    id = rpu_id++;
    int a[10];
    rpus_stack_enable(10, a, id);
    int b[10];
    rpus_stack_enable(10, b, id);
    rpus_stack_disable(id);
}
```
Proposed Debug Methodology

RPM Hardware Architecture

- Bus Adapter
- RPU controller
  - data_in
  - address
  - data_out
  - heap_in
  - stack_fallback_n
  - rpu_mode
  - read
  - write
  - rpu_data
  - stack_in
- heap RPU 1
- heap RPU N
- stack RPU 1
- stack RPU M
Proposed Debug Methodology

Heap RPU Hardware Block Diagram

cascade_in

data

clock

mode

control

base

size

sub

A<B?

or

cascade_out
**Proposed Debug Methodology**

**RPM Hardware Design Flow**

1. **Benchmark applications**
2. **Memory Usage Analysis**
3. **RPU Design Algorithm**
4. **XML description of required RPU components**
5. **IP generation & instantiation**

---

**Memory statistics**

---

Proposed Debug Methodology

Hardware Features

• Features
  – Adds fine-grain memory protection
    • Complementary to MMU’s page-based protection
  – Reconfigurable at run-time
  – Area-efficient
  – Scalable
  – Fits any (industry-)standard bus interface
    • AXI, OCP, DTL, MTL …

• Options
  – Direct bus snoop ⇔ Address sent by SW
  – Generate interrupt ⇔ Valid query in SW
  – Complementary IEEE 1149.1 (JTAG) access
Proposed Debug Methodology

Software Design Flow

- Application compile time
  - Identify regions to protect per thread using the compiler
  - Instrument application

- Application run-time
  - Memory region violations detected by RPU hardware
  - Handling is done by
    - CPU software, and/or
    - Debugger software
Proposed Debug Methodology

Software API Example

```
main()
{
    func1();
    func2();
}

func1()
{
    id = rpu_id++;
p = malloc(127);
rpus_heap_enable(127,p);
    int a[10];
rpus_stack_enable(10,a,id);
    int b[10];
rpus_stack_enable(10,b,id);
    free(p);
    a[10]=0;
rpus_stack_disable(id);
}

func2()
{
    id = rpu_id++;
    int a[10];
    int b[10];
rpus_stack_disable(id);
}
```
Experimental Results

- Modified open-source GCC compiler on Linux
- ARM Cross-compiler
- MiBench (http://www.eecs.umich.edu/mibench/)
  - Commercially representative embedded benchmarks
  - Automotive, Consumer, Network, Office, Security, and Telecommunication
- Measured:
  - Software performance drop
  - Minimum number of required RPUs
Experimental Results

Application Speed per Benchmark

Benchmark: basicmath, bitcnts, qsort, susan_corners, susan_edges, susan_smoothing, sha, crc32, fft, ifft, adpcm_c, adpcm_d, ispell, search

Application Speed (Original = 100%)

Mudflap, RPM (no snooping), RPM (snooping)
Experimental Results

RPU Hardware Cost

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Mudflap</th>
<th>RPM (no snooping)</th>
<th>RPM (snooping)</th>
</tr>
</thead>
<tbody>
<tr>
<td>basicmath</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bitcnts</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>qsort</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>susan_corners</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>susan_edges</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>susan_smoothing</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sha</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>crc32</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fft</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ifft</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>adpcm_c</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>adpcm_d</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ispell</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>search</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Conclusions

• Run-Time Memory Protection Architecture
  – Effective against memory corruption
  – Efficient through
    • Re-use of existing RPU hardware
    • Optimal trade-off between HW and SW cost

• We developed tool support for
  – Memory allocation & access analysis
  – Hardware and software trade-off
  – RPU hardware design
  – Application instrumentation
Thank You