DSA Monitoring Framework for HW/SW Partitioning of Application Kernels leveraging VPs

Christoph Hazott, Daniel Große
Institute for Complex Systems (ICS)
Web: jku.at/ics
Email: christoph.hazott@jku.at
Agenda

• Motivation, Domain Specific Architectures & Background on VPs
• Ingredients of Proposed approach
• Solution:
  ◦ Host-to-SW Memory Hierarchy
  ◦ Proposed Framework
• Experiments
• Conclusion
Motivation and DSAs

• Moore’s Law
  ◦ Still driving electronic industry
  ◦ Permanent innovation on all levels necessary to keep speed

• Domain Specific Architectures (DSAs)
  ◦ Fall into class of heterogeneous architectures
  ◦ Integrate specific HW accelerators extracted from SW to meet system performance requirements
  ◦ HW/SW partitioning through application kernels
    ▪ Hotspot functions invoked frequently in loops
A Virtual Prototype (VP) is an executable SW model of a HW system that runs on a host computer.

A VP is binary compatible to the physical HW.

- Industrial-proven, widely used by semiconductor global players
- VPs are modeled in SystemC
  - C++ class library
  - IEEE1666-2011 Standard
  - Transaction Level Modeling (TLM) for abstraction
Background: Why Virtual Prototypes?

Traditional Development Flow

Parallel HW and SW Development based on VP

Source: Synopsys
Profiling with Virtual Prototypes?

- Classical Profiling
  - HW and SW separately
  - Joined in later processing of result
  - Changes to source code or binary

- Proposed Approach
  - Taking external perspective that encompasses HW and SW, facilitating a holistic view as unified system

Source: Synopsys
Ingredients of Proposed Approach

• Framework leveraging observability of VPs
• Monitoring via runtime code manipulation of VP binary running on host
  ◦ Maintain high simulation performance
• Specialized monitors for SW kernels
  ◦ Low data amount

• Requirements:
  ◦ No modification of SW running on VP
  ◦ No modification of VP source

• Idea:
  ◦ Understand Host-to-SW memory hierarchy
Solution: Host-to-SW Memory Hierarchy

iss.c

```c
switch(op){
    case ADDI:
        rd = rs1 + imm;
        break;
    ...
    case SW: {
        addr = rs1 + imm;
        mem->store_word(
            addr, 
            rs2);
    }
    break;
    case LW: {
        addr = rs1 + imm;
        rd = mem->load_word(
            addr);
    }
    break;
    ...
}
```

canny.c

```c
for(int i = 1; i < cols; i++){
    deltaX[i] = image[i] - image[i-1];
}
```
Proposed Framework

Framework Configuration → Source Code Translator → Address Table → DSA Monitor → Monitoring Dataset → DSA Analyzer

<<Debug Information>>

DSA Monitor

DynamoRIO

HW

SW

<<Debug Information & Source Code>>
Source Code Translator

• From configuration

1  HW:'riscv_vp'
2  ...
3  PC_HW:'PC_vp'
4  ...
5  MEM_READ_HW:'/riscv_vp/memory.h:76'
6  ...

• To address table

1  PC_HW:'0xf33c88'
2  ...
3  MEM_READ_HW:'0x373ec'
4  ...

Johannes Kepler University Linz
DynamoRIO

• DynamoRIO as runtime code manipulation system
  ◦ Supports code transformation while executing
  ◦ Exports interface for building additional tools
  ◦ Powerful instruction manipulation library

• Design Goals
  ◦ Efficient
    ▪ Near-native performance
  ◦ Transparent
    ▪ Match native behavior
  ◦ Comprehensive
    ▪ Control every instruction, in any application
  ◦ Customizable
    ▪ Adapt to satisfy disparate tool needs

DSA Monitor

• Instrumenting for monitoring $PC_{HW}$

```c
1  if(instr_writes_memory(instr)){
2      addr = opnd_get_addr(
3          instr_get_dst(instruction, i)
4          );
5
6      if(addr == pc_hw) {
7          dr_insert_clean_call(
8              ..., clean_call_pc_hw, ...);
9      }
10 }
```

• Instrumenting for monitoring HW memory access

```c
1  pc_host = instr_get_app_pc(instr);
2  if (monitor_pc_host[pc_host]) {
3      dr_insert_clean_call(...,
4          clean_call_hw_mem_read, ...);
4  }
```
DSA Analyzer

- Data structure containing monitoring results used for analysis
Experiments: Canny Edge Detection

- **SW: Canny Edge Detection**
  - Gaussian *smoothing*
  - Computing *derivatives*
  - Computing *magnitude* of gradient
  - Performing non-maximal *suppression*
  - Applying *hysteresis*
SW Application Kernel: Excerpts from Canny SW

- Nested SW kernels implementing gaussian filter for smoothing image rows

```c
for (c=0; c<cols; c++) {
    for (r=0; r<rows; r++) {
        ...  
        for (rr=(-center); rr<=center; rr++) {  
            row = r + rr;  
            if (row >= 0 && row < rows) {  
                ...  
            }  
        }  
        smoothedim[r*cols+c] = ...  
    }  
}
```
RISC-V

• RISC-V: Open and royalty-free ISA
• Focus on simplicity and modularity
• Base Integer Instruction Set
  ◦ Mandatory
  ◦ 32, 64 and 128 bit configurations
  ◦ ~40 Instructions
• Extensions:
  ◦ M .. Multiply/Divide
  ◦ A .. Atomic
  ◦ F, D, Q .. Floating Point (Single, Double, Quad)
  ◦ C .. Compressed
RISC-V VP++

- Open source on GitHub
  - [https://github.com/ics-jku/riscv-vp-plusplus](https://github.com/ics-jku/riscv-vp-plusplus)

- Key features:
  - SystemC TLM-2.0
  - Bare metal configurations, including:
    - SiFive HiFive1 - FE310
    - GD32VF103VBT6 microcontroller (Nuclei N205) including UI
  - Linux RV32 and RV64, single and quad-core VPs (SiFive FE540)
  - Support for RISC-V "V" Vector Extension (RVV) version 1.0
  - Full integration of GUI-VP, which enables simulation of interactive graphical Linux applications
  - Based on RISC-V VP introduced in 2018*

- More information: [http://www.systemc-verification.org](http://www.systemc-verification.org)

Results

• Costs and Scalability of Monitoring

<table>
<thead>
<tr>
<th></th>
<th>11x9</th>
<th>22x18</th>
<th>44x36</th>
<th>88x72</th>
<th>176x144</th>
<th>352x288</th>
<th>704x576</th>
<th>1056x864</th>
</tr>
</thead>
<tbody>
<tr>
<td>Executed RISC-V instructions</td>
<td>2,918,759</td>
<td>7,350,190</td>
<td>25,637,385</td>
<td>98,984,371</td>
<td>395,274,305</td>
<td>1,584,436,030</td>
<td>6,502,028,739</td>
<td>14,746,998,363</td>
</tr>
<tr>
<td>Host time - no monitoring [min]</td>
<td>0.01</td>
<td>0.03</td>
<td>0.09</td>
<td>0.32</td>
<td>1.28</td>
<td>4.97</td>
<td>20.66</td>
<td>51.84</td>
</tr>
<tr>
<td>Host time - monitoring [min]</td>
<td>0.04</td>
<td>0.06</td>
<td>0.17</td>
<td>0.60</td>
<td>2.39</td>
<td>9.20</td>
<td>38.02</td>
<td>93.24</td>
</tr>
<tr>
<td>Overhead factor</td>
<td>3.45</td>
<td>2.19</td>
<td>1.91</td>
<td>1.84</td>
<td>1.86</td>
<td>1.85</td>
<td>1.84</td>
<td>1.80</td>
</tr>
<tr>
<td>Size of monitoring dataset [MB]</td>
<td>34</td>
<td>85</td>
<td>294</td>
<td>1,229</td>
<td>4,608</td>
<td>18,432</td>
<td>74,752</td>
<td>168,960</td>
</tr>
</tbody>
</table>

- Results scale according to the number of pixels
- Starting from a resolution of ~88x72px, overhead factor stabilizes at ~1.85
- Results stabilize at 88x72px
HW/SW Partitioning: Simulation Time

(a) Overall  
(b) RV32I  
(c) +MAFC extensions  
(d) +HW smoothing
HW/SW Partitioning: Memory Accesses

(a) Overall  (b) RV32I  (c) +MAFC extensions  (d) +HW smoothing
HW/SW Partitioning: Acceleration

• Performance improved by a factor of \(~8.67\)
Conclusions

• Novel monitoring approach
  ◦ Host-to-SW memory hierarchy
  ◦ Leveraging dynamic binary instrumentation to insert monitors
  ◦ Low simulation overhead
  ◦ Low data amount

• Framework
  ◦ User interaction via source code
  ◦ Graphs for HW/SW interactions
DSA Monitoring Framework for HW/SW Partitioning of Application Kernels leveraging VPs

Christoph Hazott, Daniel Große
Institute for Complex Systems (ICS)
Web: jku.at/ics
Email: christoph.hazott@jku.at