Operating Systems Should Manage Accelerators

Sankaralingam Panneerselvam
Michael M. Swift

Computer Sciences Department
University of Wisconsin, Madison, WI
Accelerators

- Specialized execution units for common tasks
  - Data parallel tasks, encryption, video encoding, XML parsing, network processing etc.
- Task offloaded to accelerators
  - High performance, energy efficient (or both)
Why accelerators?

- Moore’s law and Dennard’s scaling
  - Single core to Multi-core
  - Dark Silicon: More transistors but limited power budget [Esmaeilzadeh et. al - ISCA ‘11]

- Accelerators
  - Trade-off area for specialized logic
  - Performance up by 250x and energy efficiency by 500x [Hameed et. al – ISCA ’10]

- Heterogeneous architectures
Heterogeneity everywhere

- Focus on diverse heterogeneous systems
  - Processing elements with different properties in the same system

- Our work
  - How can OS support this architecture?
    1. Abstract heterogeneity
    2. Enable flexible task execution
    3. Multiplex shared accelerators among applications
Outline

- Motivation
- Classification of accelerators
- Challenges
- Our model - Rinnegan
- Conclusion
IBM wire-speed processor

CPU cores

Accelerator Complex

Memory controller

Comp/Decomp
Crypto
XML
Pattern Engine

PBus

PBIC

PLL
PLL
PLL
PLL
PLL
PLL
Classification of accelerators

- Accelerators classified based on their accessibility
  - Acceleration devices
  - Co-Processor
  - Asymmetric cores
## Classification

<table>
<thead>
<tr>
<th>Properties</th>
<th>Acceleration devices</th>
<th>Co-Processor</th>
<th>Asymmetric cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>Systems</td>
<td>GPU, APU, Crypto accelerators in Sun Niagara chips etc.</td>
<td>C-cores, DySER, IBM wire-speed processor</td>
<td>NVIDIA’s Kal-El processor, Scalable cores like WiDGET</td>
</tr>
<tr>
<td>Location</td>
<td>Off/On-chip device</td>
<td>On-chip device</td>
<td>Regular core that can execute threads</td>
</tr>
</tbody>
</table>

### Accessibility

| Data       | DMA or zero copy     | System memory | Cache coherence |

### Resource contention

1) Application must deal with different classes of accelerators with different access mechanisms
2) Resources to be shared among multiple applications

Multiple applications might want to make use of the device
Sharing of resource across multiple cores could result in contention
Multiple threads might want to access the special core
Challenges

- Task invocation
  - Execution on different processing elements

- Virtualization
  - Sharing across multiple applications

- Scheduling
  - Time multiplexing the accelerators
Task invocation

- Current systems decide statically on where to execute the task
- Programmer problems where system can help
  - Data granularity
  - Power budget limitation
  - Presence/availability of accelerator

AES

Crypto accelerator
Virtualization

- Data addressing for accelerators
  - Accelerators need to understand virtual addresses
  - OS must provide virtual-address translations
- Process preemption after launching a computation
  - System must be aware of computation completion
Scheduling

- Scheduling needed to prioritize access to shared accelerators
- User-mode access to accelerators complicates scheduling decisions
  - OS cannot interpose on every request
Outline

- Motivation
- Classification of Accelerators
- Challenges
- Our model - Rinnegan
- Conclusion
Rinnegan

- **Goals**
  - Application leverage the heterogeneity
  - OS enforces system wide policies

- Rinnegan tries to provide support for different class of accelerators
  - Flexible task execution by **Accelerator stub**
  - **Accelerator agent** for safe multiplexing
  - Enforcing system policy through **Accelerator monitor**
Accelerator stub

- Entry point for accelerator invocation and a dispatcher
- Abstracts different processing elements
- Binding - Selecting the best implementation for the task
  - Static: during compilation
  - Early: at process start
  - Late: at call

{ Encrypt through AES instructions }

{ Offload to crypto Accelerator }

Stub
Process making use of accelerator
Accelerator agent

- Manages an accelerator

Roles of an agent

- Virtualize accelerator
- Forwarding requests to the accelerator
- Implement scheduling decisions
- Expose accounting information to the OS
Accelerator monitor

- Monitors information like utilization of resources
- Responsible for system-wide energy and performance goals
- Acts as an online modeling tool
  - Helps in deciding where to schedule the task
Programming model

- Based on task level parallelism
- Task creation and execution
  - **Write**: different implementations
  - **Launch**: similar to function invocation
  - **Fetching result**: supporting synchronous/asynchronous invocation
- Example: Intel TBB/Cilk

```cpp
application()
{
    task = cryptStub(inputData, outputData);
    ...
    waitFor(task);
}
<TaskHandle> cryptStub(input, output)
{
    if(factor1 == true)
        output = implmn_AES(input)
        //or//
    else if(factor2 == true)
        output = implmn_CACC(input)
}
```
Summary

Accelerator agent

Process making use of accelerator

CPU cores

Accelerator

Other kernel components

Accelerator monitor

Process

Kernel Space

User Space

Direct communication with accelerator
Related work

- Task invocation based on data granularity
  - Merge, Harmony
- Generating device specific implementation of same task
  - GPUOcelot
- Resource scheduling: Sharing between multiple applications
  - PTask, Pegasus
Conclusions

- Accelerators common in future systems
- Rinnegan embraces accelerators through stub mechanism, agent and monitoring components

Other challenges
- Data movement between devices
- Power measurement
Thank You