PERFORMANCE MONITORING OF ARCHITECTURALLY DIVERSE SYSTEMS

Joseph M. Lancaster, Roger D. Chamberlain
Dept. of Computer Science and Engineering
Washington University in St. Louis
{lancaster, roger}@wustl.edu

Research supported by NSF grant CNS-0720667
We want apps that...

- Run correctly
  - Do not dead-lock
  - Meet hard real-time deadlines
- Run fast
  - High-throughput / low latency
  - Low rate of soft deadline misses

Infrastructure should help us debug when it runs incorrectly or slow
Diverse Computing

- Increasingly common in HPEC systems
- e.g. Mercury, XtremeData, DRC, Nallatech, ClearSpeed

![Diagram showing CMP and FPGA components]
App deployed using all four components
Diverse Computing
Benefits and Challenges

- Large performance gains realized
- Power efficient compared to CMP alone

- Requires knowledge of individual architectures/languages
- Components operate independently
  - Distributed system
  - Separate memories and clocks
Motivation

Tool support for these systems insufficient

- Many architectures lack tools for monitoring and validation
- Tools for different architectures not integrated
- Ad hoc solutions

Solution: Runtime performance monitoring and validation for diverse systems!
Outline

- Introduction
- Runtime performance monitoring
- Frame monitoring
- User-guidance
Stream Programming

- Natural fit for diverse HPEC systems
- Dataflow model
  - Composed of blocks and edges
  - Blocks compute concurrently
  - Data flows along edges
- Languages: StreamIt, Streams-C, X
Mapping Process

A → B → D

C

CMP

CORE 1

A

CORE 2

D

FPGA

B

GPU

C
## Existing Performance Tools

<table>
<thead>
<tr>
<th>Programming model</th>
<th>Strategy</th>
<th>Tools / Environments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shared Memory</td>
<td>Execution profiling</td>
<td>gprof, Valgrind, PAPI</td>
</tr>
<tr>
<td>Message Passing</td>
<td>Execution profiling, message logging</td>
<td>TAU, mpiP, PARAVER</td>
</tr>
<tr>
<td>Stream Programming</td>
<td>Simulation</td>
<td>StreamIt [MIT], StreamC [Stanford], Streams-C [LANL], Auto-Pipe [WUSTL]</td>
</tr>
</tbody>
</table>
Traditional Performance Monitoring

- Limitations for diverse systems
  - No universal PC or architecture
  - No shared memory
  - Different clocks
  - Communication latency and bandwidth
Diverse System Performance Monitoring

- Simulation is a useful first step but:
  - Models can abstract away system details
  - Too slow for large datasets
  - HPEC applications growing in complexity

- Need to monitor deployed, running app
  - Measure *actual* performance of system
  - Validate performance of *large, real-world* datasets
Goals

- Report more than just aggregate statistics
  - Capture rare events
- Quantify measurement impact where possible
  - Overhead due to sampling, communication, etc.
- Measure runtime performance efficiently
  - Low overhead
  - High accuracy
- Validate performance of real datasets
- Increase developer productivity
Monitoring Strategy

- Monitor edges / queues
- Find bottlenecks in app
  - Change over time?
  - Computation or communication?
- Measure latency between two points
Interconnects are a precious resource
Uses same interconnects as application
Stay below bandwidth constraint → Keep perturbation low
Reducing Impact

- Understand measurement perturbation
- Dedicate compute resources when possible
- Aggressively reduce amount of performance meta-data stored and transmitted
  - Utilize compression in both time resolution and fidelity of data values
  - Use knowledge from user to specify their performance expectations / measurements
Dedicating Resources

- Use CMP core as the server monitor
  - Monitor other cores for performance information
  - Process data from agents (e.g. FPGA, GPU)
  - Combine hardware and software information for global view
    - Use logical clocks to synchronize events
- Dedicate unused FPGA area to monitoring
Outline

- Introduction
- Runtime Performance Monitoring
  - Frame monitoring
  - User-guidance
Temporal Monitoring Continuum

Unbounded Monitoring bandwidth

Complete fidelity

Temporal or precision tradeoffs

Aggregate statistics

No Monitoring Bandwidth
Frame Monitoring

- A frame summarizes performance over a period of the execution
- Maintain some temporal information
  - Capture system performance anomalies
Frame Monitoring

- A frame summarizes performance over a period of the execution
- Maintain some temporal information
  - Capture system performance anomalies
Frame Monitoring

- A frame summarizes performance over a period of the execution
- Maintain some temporal information
  - Capture system performance anomalies
Frame Monitoring

- A frame summarizes performance over a period of the execution
- Maintain some temporal information
  - Capture system performance anomalies

[Diagram showing timeframes with performance metrics]

Time

1 2 3 4 5 6 7 8 9
Properties of Frames

- Each frame reports one performance metric
- Frame size can be dynamic
  - Dynamic bandwidth budget
  - Low variance data / application phases
  - Trade temporal granularity for lower perturbation
- Frames from different agents will likely be unsynchronized and different sizes
- Monitor server presents user with consistent global view of performance
Outline

- Introduction
- Runtime Performance Monitoring
- Frame Monitoring
- User-guidance
User-guided Collection

- **Why?**

- **Related work: Performance Assertions for Mobile Devices [Lenecevicius’06]**
  - Validates user performance assertions on multi-threaded embedded CPU

- Our system enables validation of performance expectations **across diverse architectures**
User-Guided Collection

1. Measurement
   - User specifies a set of “taps” for agent
   - Taps can be off an edge or an input queue
   - Agent then records events on each tap
   - Supported measurements for a tap:
     - Average value + standard deviation
     - Min or max value
     - Histogram of values
     - Outliers (based on parameter)
   - Basic arithmetic and logical operators on taps:
     - Arithmetic: add, subtract, multiply, divide
     - Logic: and, or, not
Direct Measurement Example

What is the throughput of block A?
What is throughput of block A when it is not data starved?
Direct Measurement Example

What is the throughput of block A when
- not starved for data and
- no downstream congestion
User-Guided Collection (2)

1. Measurement
   - Set of “taps” for agent to count, histogram, or perform simple logical operations on
   - Taps can be an edge or an input queue

2. Performance assertion
   - User describes their performance expectations of an application as assertions
   - Runtime monitor validates these assertions by collecting measurements and evaluating logical expressions
     - Arithmetic operators: +, -, *, /
     - Logical operators: and, or, not
     - Annotations: t, L
Logic of Constraints

- **throughput:** “at least 100 $A.Input$ events will be produced in any period of 1001 time units”
  - $t(A.Input[i+100]) - t(A.Input[i]) \leq 1001$

- **latency:** “$A.Output$ is generated no more than 125 time units after $A.Input$”
  - $t(A.Output[i]) - t(A.Input[i]) \leq 125$

- **queue bound:** “$A.InQueue$ never exceeds 100 elements”
  - $L(A.InQueue[i]) \leq 100$
Abstraction Hierarchy

- Runtime measurements
  - Query CMP/GPU performance counters
  - Custom FPGA counters

- Local assertions
  - Can be evaluated within a single agent
  - No need for communication with other agents/system monitor

- Global assertions
  - Requires aggregating results from more than one agent on different compute resources
Constraint Properties

- Some assertions impose prohibitive memory requirements
  - Either disallow these or warn user of large monitoring impact
- Other assertions are compute intensive
- A few are both!
- Fortunately, much can be gained from simple queries
  - Input queue lengths over time
Status

- FPGA Agent mostly operational
  - Monitor only, no user assertions yet
- Initial target application is the BLAST biosequence analysis application
  - CPU + FPGA hardware platform
  - [Jacob, et al. TRETS ’08]
- Next target application is computational finance
  - CPU + GPU + FPGA
  - Performance significantly worse than models
Conclusions & Future Work

- Runtime performance monitoring enables
  - More efficient development
  - Better testing for real-time systems

- Support correctness assertions
- Investigate ways to best present results to developer