Ubiquitous Memory Introspection (UMI)

Qin Zhao, NUS
Rodric Rabbah, IBM
Saman Amarasinghe, MIT
Larry Rudolph, MIT
Weng-Fai Wong, NUS

CGO 2007, March 14 2007
San Jose, CA
The complexity crunch

hardware complexity ↑
+ 
software complexity ↑
↓
too many things happening concurrently, complex interactions and side-effects
down
less understanding of program execution behavior ↓
The importance of program behavior characterization

Better understanding of program characteristics can lead to more robust software, and more efficient hardware and systems.

- **S/W developer**
  - Computation vs. communication ratio
  - Function/path frequency
  - Test coverage

- **H/W designer**
  - Cache behavior
  - Branch prediction

- **System administrator**
  - Interaction between processes

Better understanding of program characteristics can lead to more robust software, and more efficient hardware and systems.
<table>
<thead>
<tr>
<th>Common approaches to program understanding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overhead</td>
</tr>
<tr>
<td>Level of detail</td>
</tr>
<tr>
<td>Versatility</td>
</tr>
</tbody>
</table>
### Common approaches to program understanding

<table>
<thead>
<tr>
<th></th>
<th>Profiling</th>
<th>Simulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overhead</td>
<td>high</td>
<td>very high</td>
</tr>
<tr>
<td>Level of detail</td>
<td>very high</td>
<td>very high</td>
</tr>
<tr>
<td>Versatility</td>
<td>high</td>
<td>very high</td>
</tr>
</tbody>
</table>
## Common approaches to program understanding

<table>
<thead>
<tr>
<th></th>
<th>Profiling</th>
<th>Simulation</th>
<th>HW Counters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overhead</td>
<td>high</td>
<td>very high</td>
<td>very low</td>
</tr>
<tr>
<td>Level of detail</td>
<td>very high</td>
<td>very high</td>
<td>very low</td>
</tr>
<tr>
<td>Versatility</td>
<td>high</td>
<td>very high</td>
<td>very low</td>
</tr>
</tbody>
</table>
Slowdown due to HW counters
(counting L1 misses for 181.mcf on P4)

(% slowdown)
### Common approaches to program understanding

<table>
<thead>
<tr>
<th></th>
<th>Profiling</th>
<th>Simulation</th>
<th>HW Counters</th>
<th>Desirable Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overhead</td>
<td>high</td>
<td>very high</td>
<td>very low</td>
<td>low</td>
</tr>
<tr>
<td>Level of detail</td>
<td>very high</td>
<td>very high</td>
<td>very low</td>
<td>high</td>
</tr>
<tr>
<td>Versatility</td>
<td>high</td>
<td>very high</td>
<td>very low</td>
<td>high</td>
</tr>
</tbody>
</table>
Common approaches to program understanding

<table>
<thead>
<tr>
<th></th>
<th>Profiling</th>
<th>Simulation</th>
<th>HW Counters</th>
<th>UMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overhead</td>
<td>high</td>
<td>very high</td>
<td>very low</td>
<td>low</td>
</tr>
<tr>
<td>Level of detail</td>
<td>very high</td>
<td>very high</td>
<td>very low</td>
<td>high</td>
</tr>
<tr>
<td>Versatility</td>
<td>high</td>
<td>very high</td>
<td>very low</td>
<td>high</td>
</tr>
</tbody>
</table>
Key components

• Dynamic Binary Instrumentation
  - Complete coverage, transparent, language independent, versatile, ...

• Bursty Simulation
  - Sampling and fast forwarding techniques
  - Detailed context information
  - Reasonable extrapolation and prediction
Ubiquitous Memory Introspection

online mini-simulations analyze short memory access profiles recorded from frequently executed code regions

• Key concepts
  - Focus on hot code regions
  - Selectively instrument instructions
  - Fast online mini-simulations
  - Actionable profiling results for online memory optimizations
Working prototype

- Implemented in DynamoRIO
  - Runs on Linux
  - Used on Intel P4 and AMD K7

- Benchmarks
  - SPEC 2000, SPEC 2006, Olden
  - Server apps: MySQL, Apache
  - Desktop apps: Acrobat-reader, MEncoder
UMI is cheap and non-intrusive (SPEC2K reference workloads on P4)

- Average overhead is 14%
- 1% more than DynamoRIO
What can UMI do for you?

• Inexpensive introspection everywhere

• **Coarse grained memory analysis**
  - Quick and dirty

• **Fine grained memory analysis**
  - Expose opportunities for optimizations

• **Runtime memory-specific optimizations**
  - Pluggable prefetching, learning, adaptation
Coarse grained memory analysis

• Experiment: measure cache misses in three ways
  - HW counters
  - Full cache simulator (Cachegrind)
  - UMI

• Report correlation between measurements
  - Linear relationship of two sets of data
Cache miss correlation results

- **HW counter vs. CacheGrind**
  - 0.99
  - 20x to 100x slowdown

- **HW counter vs. UMI**
  - 0.88
  - Less than 2x slowdown in worst case
What can UMI do for you?

• Inexpensive introspection everywhere

• Coarse grained memory analysis
  - Quick and dirty

• Fine grained memory analysis
  - Expose opportunities for optimizations

• Runtime memory-specific optimizations
  - Pluggable prefetching, learning, adaptation
Fine grained memory analysis

- Experiment: predict delinquent loads using UMI
  - Individual loads with cache miss rate greater than threshold
- Delinquent load set determined according to full cache simulator (Cachegrind)
  - Loads that contribute 90% of total cache misses
- Measure and report two metrics
  - Delinquent loads identified by Cachegrind
  - Delinquent loads predicted by UMI recall
    - false positive
<table>
<thead>
<tr>
<th></th>
<th>Recall (higher is better)</th>
<th>False positive (lower is better)</th>
</tr>
</thead>
<tbody>
<tr>
<td>benchmarks with ≥ 1% miss rate</td>
<td>88%</td>
<td>55%</td>
</tr>
<tr>
<td>benchmarks with &lt; 1% miss rate</td>
<td>26%</td>
<td>59%</td>
</tr>
</tbody>
</table>
What can UMI do for you?

• Inexpensive introspection everywhere

• Coarse grained memory analysis
  - Quick and dirty

• Fine grained memory analysis
  - Expose opportunities for optimizations

• Runtime memory-specific optimizations
  - Pluggable prefetcher, learning, adaptation
Experiment: online stride prefetching

- Use results of delinquent load prediction
- Discover stride patterns for delinquent loads
- Insert instructions to prefetch data to L2
- Compare runtime for UMI and P4 with HW stride prefetcher
Data prefetching results summary

Running time normalized to native execution (lower is better)

- SW prefetching
- HW prefetching
- Combined
The Gory Details
UMI components

- Region selector
- Instrumentor
- Profile analyzer
Region selector

• Identify representative code regions
  – Focus on traces, loops
  – Frequently executed code
  – Piggy back on binary instrumentor tricks

• Reinforce with sampling
  – Time based, or leverage HW counters
  – Naturally adapt to program phases
Instrumentor

• Record address references
  – Insert instructions to record address referenced by memory operation

• Manage profiling overhead
  – Clone code trace (akin to Arnold-Ryder scheme)
  – Selective instrumentation of memory operations
    • E.g., ignore stack and static data
Recording profiles

**Code Trace Profile**

- T1
- T1
- T2
- T1

Page protection to detect profile overflow

**Address Profiles**

### Code Trace T1

<table>
<thead>
<tr>
<th>op1</th>
<th>op2</th>
<th>op3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x011</td>
<td>0x024</td>
<td>0x100</td>
</tr>
<tr>
<td>0x012</td>
<td>early trace exist</td>
<td>early trace exist</td>
</tr>
<tr>
<td>0x013</td>
<td>0x028</td>
<td>0x104</td>
</tr>
</tbody>
</table>

### Code Trace T2

<table>
<thead>
<tr>
<th>op1</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x032</td>
<td>0x031</td>
</tr>
</tbody>
</table>

(counter)
Mini-simulator

• Triggered when code or address profile is full

• Simple cache simulator
  - Currently simulate L2 cache of host
  - LRU replacement
  - Improve approximations with techniques similar to offline fast forwarding simulators
    • Warm up and periodic flushing

• Other possible analyzer
  - Reference affinity model
  - Data reuse and locality analysis
Mini-simulations and parameter sensitivity

181.mcf

- Regular data structures
- If sampling threshold too high, starts to exceed loop bounds: miss out on profiling important loops
- Adaptive threshold is best
Mini-simulations and parameter sensitivity

- Irregular data structures
- Need longer profiles to reach useful conclusions
Summary

• UMI is lightweight and has a low overhead
  - 1% more than DynamoRIO
  - Can be done with Pin, Valgrind, etc.
  - No added hardware necessary
  - No synchronization or syscall headaches
  - Other cores can do real work!

• Practical for extracting detailed information
  - Online and workload specific
  - Instruction level memory reference profiles
  - Versatile and user-programmable analysis

• Facilitate migration of offline memory optimizations to online setting
Future work

• More types of online analysis
  - Include global information
  - Incremental (leverage previous execution info)
  - Combine analysis across multiple threads

• More runtime optimizations
  - E.g., advanced prefetch optimization
    • Hot data stream prefetch
    • Markov prefetcher
  - Locality enhancing data reorganization
    • Pool allocation
    • Cooperative allocation between different threads

• Your ideas here...