Inspector

Data Provenance using Intel Processor Trace (PT)

Jörg Thalheim
Pramod Bhatotia & Christof Fetzer

Technical University of Dresden
Data Provenance

...records transformations made on data to explain how the computation was performed
Motivation: Use-case examples

Dependability:
  - Debugging programs

Security:
  - Dynamic Information Flow Tracking (DIFT)

Efficiency:
  - Memory management for NUMA
Research gap

- Currently limited either to sequential programs

For parallel programs:

- Require manual annotations w/ new type systems
- Restrictive programming model & synchronization primitives
Design goals

● Transparency
  ○ Unmodified multithreaded programs

● Generality
  ○ Shared-memory model w/ POSIX sync. primitives

● Efficiency
  ○ Low overheads using a parallel provenance algorithm
Inspector: Easy to use!

$ inspector -- ./<program> <arguments>

1. Preloads the Inspector library
   LD_PRELOAD=libinspector.so
2. Executes “existing binaries” w/o re-compilation
3. Writes the provenance log to ./perf.data
Agenda

- √ Motivation
- Design
- Implementation
- Evaluation
Behind the scenes

**step#1**
Divide

**step#2**
Read/write set

<table>
<thead>
<tr>
<th>Read</th>
<th>Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>{x,y}</td>
<td>{x}</td>
</tr>
<tr>
<td>{y}</td>
<td>{}</td>
</tr>
</tbody>
</table>

Read/write sets

**step#3**
Dependencies

Provenance graph
A simple example

Shared variables: x and y

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>lock();</code></td>
<td><code>lock();</code></td>
</tr>
<tr>
<td>if (flag ==0)</td>
<td></td>
</tr>
<tr>
<td>x = ++y;</td>
<td>y = 2 * x;</td>
</tr>
<tr>
<td>else</td>
<td></td>
</tr>
<tr>
<td>x = (++y) + 5;</td>
<td></td>
</tr>
<tr>
<td><code>unlock();</code></td>
<td><code>unlock();</code></td>
</tr>
<tr>
<td><code>lock();</code></td>
<td></td>
</tr>
<tr>
<td>y = y/2;</td>
<td></td>
</tr>
<tr>
<td><code>unlock();</code></td>
<td></td>
</tr>
</tbody>
</table>
Step #1: Sub-computations

Shared variables: x and y

Thread 1

lock();
if (flag == 0)
x = ++y;
else
x = (++y) + 5;
unlock();

Thread 2

lock();
y = y/2;
unlock();

lock();
y = 2 * x;
unlock();

Approaches:

Coarsed Grained
○ Whole Thread
→ imprecise

Fine Grained
○ Every Instruction
→ expensive

Middle Ground
○ Sub-Computations
Step #2: Read-write sets

Shared variables: x and y

Thread 1

- `lock();`
- `if (flag == 0)`
  - `x = ++y;`
  - `else`
  - `x = (++y) + 5;`
- `unlock();`
- `lock();`
- `y = y/2;`
- `unlock();`

Thread 2

- `lock();`
- `y = 2 * x;`
- `unlock();`

Read = {y};
Write = {x, y};

Read = {x};
Write = {y};
Step #3: Provenance graph

We record three dependencies:

A. Control
B. Schedule
C. Data
A: Control dependencies

Shared variables: x and y

Thread 1

lock();
if (flag == 0)
  x = ++y;
else
  x = (++y) + 5;
unlock();

lock();
y = y/2;
unlock();

Thread 2

lock();
y = 2 * x;
unlock();
B: Synchronization dependencies

Shared variables: x and y

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>lock();</code></td>
<td><code>lock();</code></td>
</tr>
<tr>
<td>if (flag ==0)</td>
<td>y = 2 * x;</td>
</tr>
<tr>
<td>x = ++y;</td>
<td><code>unlock();</code></td>
</tr>
<tr>
<td>else</td>
<td><code>lock();</code></td>
</tr>
<tr>
<td>x = (++y) + 5;</td>
<td>y = 2 * x;</td>
</tr>
<tr>
<td><code>unlock();</code></td>
<td><code>unlock();</code></td>
</tr>
<tr>
<td><code>lock();</code></td>
<td><code>lock();</code></td>
</tr>
<tr>
<td>y = y/2;</td>
<td>y = 2 * x;</td>
</tr>
<tr>
<td><code>unlock();</code></td>
<td><code>unlock();</code></td>
</tr>
<tr>
<td><code>lock();</code></td>
<td><code>lock();</code></td>
</tr>
<tr>
<td>y = y/2;</td>
<td>y = 2 * x;</td>
</tr>
<tr>
<td><code>unlock();</code></td>
<td><code>unlock();</code></td>
</tr>
</tbody>
</table>

Change of Schedule
**C: Data dependencies**

Shared variables: x and y

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>lock();</strong></td>
<td><strong>lock();</strong></td>
</tr>
<tr>
<td>if (flag == 0)</td>
<td>y = 2 * x;</td>
</tr>
<tr>
<td>x = ++y;</td>
<td><strong>lock();</strong></td>
</tr>
<tr>
<td>else</td>
<td>y = y/2;</td>
</tr>
<tr>
<td>x = (++y) + 5;</td>
<td><strong>unlock();</strong></td>
</tr>
<tr>
<td><strong>unlock();</strong></td>
<td></td>
</tr>
<tr>
<td>read = {y};</td>
<td>read = {x};</td>
</tr>
<tr>
<td>write = {x,y};</td>
<td>write = {y};</td>
</tr>
<tr>
<td>read = {y};</td>
<td></td>
</tr>
<tr>
<td>write = {y};</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note: The diagram shows the sequence of operations and the dependencies between the threads. The operations within each thread are executed atomically, ensuring the integrity of the shared variables x and y.
Concurrent Provenance Graph (CPG)

Shared variables: x and y

Thread 1:
- lock();
- if (flag == 0)
  - x = ++y;
- else
  - x = (++y) + 5;
- unlock();

Thread 2:
- lock();
- y = 2 * x;
- unlock();

Edges: Dependencies
1. Control edges
2. Synchronization edges
3. Data-dependence edges

Vertices: Sub-Computations

read = \{y\};
write = \{x, y\};

read = \{y\};
write = \{y\};

read = \{x\};
write = \{y\};
Agenda

✓ Motivation
✓ Design
☒ Implementation
☒ Evaluation
Inspector architecture

Application (unmodified executable)

Inspector

Threading library
(Data & synchronization dependencies)

OS support for Intel PT
(Control dependency)

MMU

Intel PT

Perf

Provenance graph (CPG)
Agenda

✓ Motivation
✓ Design
✓ Implementation
☑ Evaluation
Evaluation

Questions:

1. Performance overheads
2. Sources for these overheads

Experimental setup:

- Benchmarks: Phoenix 2.0 and PARSEC 3.0
- Platform: Intel Broadwell CPU with 8 cores (16 hyper-threads)

More results in the paper
Q1: Performance overheads
Q2: Source of the overheads
Inspector: Data provenance using Intel Processor Trace (PT)

- **Transparent:** Targets unmodified multithreaded programs
- **General:** Supports the shared-memory model w/ POSIX sync primitives
- **Efficient:** Employs a parallel provenance algorithm

Usage: A dynamically linkable shared library

- **Source Code:** [https://github.com/Mic92/inspector](https://github.com/Mic92/inspector)