A Case for an Interleaving Constrained Shared-Memory Multiprocessor

Presenters: Yixiang Wu, Shiyu Wang
Outline

Motivation
Background and prior work
Encoding and enforcing tested interleavings
Architectural support
Result
Conclusion
Discussion
Motivation

Parallel Programming is hard

- Verifying properties for even small code regions is NP-Complete
- It is impractical to test and verify properties for all legal interleavings
Background and Prior work

Data Race Detectors
- Happens-before based
- Lockset based

Atomicity Violation Detectors
- AVIO
Data Race Detectors (Happened Before)

Mechanism

- can detect Data Race
- re-execute the program using checkpoint and rollback.
Data Race Detectors (Happened Before) Cont.

Limitation:

- Not all data races are harmful data races
- It is heuristic
- A number of concurrency bugs cannot be detected by data race detector
Atomicity Violation Detectors (AVIO)

Automatically infer atomic regions and detect atomicity violations using dynamic analysis

- Infer atomicity invariants from the training runs
- An invariant pair cannot interleave with an unserializable memory operation accessing the same location in a different thread
Atomicity Violation Detectors (AVIO) Cont.

**Limitation**

Cannot detect all of the atomic violations
An example of bug

Neither a data race
Nor an atomicity violation
Programmers test all legal interleavings

Encode the interleaving constraints generated from test result into program binary code

Avoid many untested interleaving.
Solution (Challenges)

- How to encode the set of all tested thread interleavings in a program’s executable?
  - Deriving and Encoding Predecessor Set (PSet) constraints

- How to efficiently enforce PSet constraints during production runs in order to avoid untested interleaving?
  - Detecting and Enforcing PSet Constraints
  - Architecture support
PSet Constraints

Predecessor Sets (PSet)

- PSet is defined for each static memory operation
- PSet(m), if m is immediately memory-dependent on a remote memory operation P in at least one tested execution.
- PSets are constructed from the test runs using a profiling tool
Detecting and Enforcing PSet Constraints

Violation Detect:

M immediately dependent on P, \( P \notin \text{PSet}(M) \) → Violation Detected

Violation Recovery:

- Stall violating memory operation until the violation gets resolved
- Re-executing the program from an earlier checkpoint with an alternative thread interleaving
Limitation

- Not account for the interleavings between two or more memory operations accessing different memory locations

- Context insensitive
Architectural Support

Why?

- Runtime overhead is huge (100X or 200X)
- Binary instrument is very slow, because they use Pin

How?

- Store coherence reply message along with last writer or reader information belongs to a different thread in private cache of processor core
- Use DISE (Dynamic Instruction Stream Editing, this enables they to add customization functionality) for efficiently executing the PSet check for every memory operation
Results

A majority of the concurrency bugs are avoidable
- Data races, atomicity violations, and also other violations

Performance overhead is very low
- Untested Interleavings in well tested program rarely exists
- Less than 5% of static memory operation have a PSet size greater than one
- Processor support
Bug Avoidance Capability

Pset: Detects 15 of 17 bugs, except #16, #17

AVIO: Only Detects 6 of 7 atomicity

Happen before: Detects all bugs except 5 data race.
Learning PSets Constraints
PSet Constraint Violations in Bug Free Executions

<table>
<thead>
<tr>
<th>Programs</th>
<th>Stall</th>
<th>Rollback</th>
<th>Cannot Resolve</th>
<th>Total PSet Constraint Violations</th>
<th>Inst. Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbzip2</td>
<td>1</td>
<td>5</td>
<td>0</td>
<td>6</td>
<td>1.3E+9</td>
</tr>
<tr>
<td>aget</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1.1E+7</td>
</tr>
<tr>
<td>pfscan</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>3</td>
<td>7.4E+7</td>
</tr>
<tr>
<td>apache</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>5</td>
<td>2.8E+8</td>
</tr>
<tr>
<td>mysql</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>9.7E+8</td>
</tr>
<tr>
<td>fft</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2.3E+8</td>
</tr>
<tr>
<td>fmm</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1.6E+9</td>
</tr>
<tr>
<td>lu</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1.6E+8</td>
</tr>
<tr>
<td>radix</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>6.4E+7</td>
</tr>
<tr>
<td>blackscholes</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>8.1E+8</td>
</tr>
<tr>
<td>canneal</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>7.0E+9</td>
</tr>
</tbody>
</table>
Memory Space Overhead
Conclusion

- Testing and verifying a multithreaded program is more difficult than a single thread

- This paper makes the first step towards designing an interleaving constrained multiprocessor

- Author analyzed several bugs in real applications, and showed that the proposed system can avoid not only data races and atomicity violations, but also other unstructured memory order related bugs
Discussion

1. The approach requires programmers to run their programs hundreds of times until sufficiently large number of interleaving are exposed. Do you think this is a reasonable requirement to have?

2. Would this approach scale on massively parallel systems that have large number of threads?

3. Only ~5% of static memory operations in the benchmarks have PSet size greater than one. Does this always translate to low overhead on the dynamic execution?

4. Is it possible to extend PSet constraints to avoid multi-variable atomicity