Optimistic Shared Memory Dependence Tracing

Yanyan Jiang\textsuperscript{1}, Du Li\textsuperscript{2}, Chang Xu\textsuperscript{1}, Xiaoxing Ma\textsuperscript{1} and Jian Lu\textsuperscript{1}

\textsuperscript{1}Nanjing University
\textsuperscript{2}Carnegie Mellon University
Understanding Non-determinism

- Concurrent programs are non-deterministic
- transient behavior on specific interleaving/schedule

MySQL crashes due to null pointer dereferencing.

Source:
```
Thread 1
if (thd->proc_info) {
  fputs(thd->proc_info, ...);
}

Thread 2
thd->proc_info = NULL;
```

Understanding order of shared memory accesses

---

1 MySQL crashes due to null pointer dereferencing.
Shared Memory Dependences

- The order between consecutive accesses of a shared location

\[
\text{Thread 1} \\
\text{read(thd->proc_info)} \quad \text{write-after-read} \quad \text{write(thd->proc_info, NULL)} \\
\text{read(thd->proc_info)} \quad \text{read-after-write}
\]

- Four types of shared memory dependences
  - read-after-read (RAR), read-after-write (RAW)
  - write-after-read (WAR), write-after-write (WAW)
Using Shared Memory Dependences

- Shared memory dependence in terms of replaying
  - RAW + WAR + WAW $\rightarrow$ trivial deterministic replay
  - RAW + WAW $\rightarrow O(n)$ deterministic replay\(^1\)
  - RAW only $\rightarrow O(exp(n))$ deterministic replay\(^2\)
- Predictive trace analyses
- Data race / atomicity violation detection

---
Capturing Shared Memory Dependences

Generally, we update metadata at shared memory accesses, and check it afterwards.

- **Write**($x$)
  - I wrote $x$

- **Read**($x$)
  - anyone written $x$?

We must force atomic access!
Capturing Shared Memory Dependences: Overhead

- Program instrumentation inevitably brings overhead
- Scalable overhead: thread-local operations
  - Metadata bookkeeping
  - Extra call and branch instructions
- Non-scalable overhead: serialization
  - Forcing atomicity is much more costly
Reducing Overhead: Exploiting Thread Locality

- A thread tends to have exclusive/shared access of a shared location for a consecutive time period
- example: classical MESI cache coherence protocol

Fast path: exclusive/shared (CHEAP)
Slow path: bus traffic (EXPENSIVE)
Octet: Optimistic Tracing via Biased Locking

- Eager acquisition, lazy release

Octet: Drawback

- Biased lock is too optimistic
- thread-local accesses are indeed fast, but
- inter-thread coordination is too costly
- under write-heavy workloads it is even worse than simple locking

Can we simultaneously achieve FAST fast paths and NOT-SLOW slow paths?

\[1\] M Cao, et al. Drinking from both glasses: adaptively combining pessimistic and optimistic synchronization for efficient parallel runtime support. In WoDet, 2014.
RWTrace: Overview of the Trade-off

- Not only thread locality, but also *reads dominate writes*
- slower thread-local writes
- faster cross-thread coordinations

<table>
<thead>
<tr>
<th></th>
<th>LEAP(^1)</th>
<th>Octet</th>
<th>RWTrace</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread-local Read</td>
<td>Slow</td>
<td>Fast</td>
<td>Fast</td>
</tr>
<tr>
<td>Shared Read</td>
<td>Slow</td>
<td>Fast</td>
<td>Fast</td>
</tr>
<tr>
<td>Inter-thread Read</td>
<td>Slow</td>
<td>Very Slow</td>
<td>Slow</td>
</tr>
<tr>
<td>Thread-local Write</td>
<td>Slow</td>
<td>Fast</td>
<td>Slow</td>
</tr>
<tr>
<td>Inter-thread Write</td>
<td>Slow</td>
<td>Very Slow</td>
<td>Slow</td>
</tr>
</tbody>
</table>

1 - Serializing Writes

- Use simple mutex lock to protect writes
- each address is associated with a lock
- write-after-write dependences can be captured

<table>
<thead>
<tr>
<th></th>
<th>Octet</th>
<th>RWTrace</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread-local Read</td>
<td>Fast</td>
<td>Fast</td>
</tr>
<tr>
<td>Shared Read</td>
<td>Fast</td>
<td>Fast</td>
</tr>
<tr>
<td>Inter-thread Read</td>
<td>Very Slow</td>
<td>Slow</td>
</tr>
<tr>
<td>Thread-local Write</td>
<td>Fast</td>
<td>Slow</td>
</tr>
<tr>
<td>Inter-thread Write</td>
<td>Very Slow</td>
<td>Slow</td>
</tr>
</tbody>
</table>
1 – Serializing Writes: Performance

- Write-write data race is NOT expected
- often leads to unexpected behaviors
- developers eliminate them by synchronization

Corollary: write-time locking scales as the non-instrumented program scales!

- serializing all writes by a global lock does not hurt performance too much¹

The vast majority, as fast as possible
\(O(1)\), zero synchronization (wait-free), scalable
tests whether a thread is reading a value from previously unknown source

<table>
<thead>
<tr>
<th></th>
<th>Octet</th>
<th>RWTrace</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread-local Read</td>
<td>Fast</td>
<td>Fast</td>
</tr>
<tr>
<td>Shared Read</td>
<td>Fast</td>
<td>Fast</td>
</tr>
<tr>
<td>Inter-thread Read</td>
<td>Very Slow</td>
<td>Slow</td>
</tr>
<tr>
<td>Thread-local Write</td>
<td>Fast</td>
<td>Slow</td>
</tr>
<tr>
<td>Inter-thread Write</td>
<td>Very Slow</td>
<td>Slow</td>
</tr>
</tbody>
</table>
2 - Read Fast Path: Am I Thread-local?

- Single-sided error: may unnecessarily fall back to slow path, but never miss any read-after-write dependence

\[ V(x) = \text{tid}() \]

\[ \text{Write}(x) \]

\[ \text{Read}(x) \]

\[ \text{Verify}(V(x)) \]
3 – Read Slow Path

- Read fast path fails → fall back to slow path
- ignore the previously read value
- read again with lock (exact read-after-write dependence)

<table>
<thead>
<tr>
<th></th>
<th>Octet</th>
<th>RWTrace</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread-local Read</td>
<td>Fast</td>
<td>Fast</td>
</tr>
<tr>
<td>Shared Read</td>
<td>Fast</td>
<td>Fast</td>
</tr>
<tr>
<td>Inter-thread Read</td>
<td>Very Slow</td>
<td>Slow</td>
</tr>
<tr>
<td>Thread-local Write</td>
<td>Fast</td>
<td>Slow</td>
</tr>
<tr>
<td>Inter-thread Write</td>
<td>Very Slow</td>
<td>Slow</td>
</tr>
</tbody>
</table>
3 - Read Slow Path: Retry

V[x] = tid()
Write[x]

dependence detected
read-after-write dependence (precise)

Fast Path (detection)
Read[x]
Read(V[x])
discard

Slow Path (retry)
Read[x]
Read(V[x])
Read(V[x])
4 - Write-after-read-read Dependences

- Write-after-read dependences are many-to-one

![Diagram showing write-after-read dependences]

Read[x] \rightarrow Write[x] \rightarrow Read[x]
4 - Write-after-read Dependences: The Memory Order Trick Again, Inversely

- Maintain a read set to preserve wait-free read fast path
- no extra synchronization, still **scalable**

\[
\text{rd}(x) = \text{rd}(x) \cup \text{tid}()
\]

\[
\text{Read}(x)
\]

\[
\text{Write}(V(x))
\]

\[
\text{rd}(x) \rightarrow \text{tid}()
\]

write-after-read dependence
There can be tricky memory orderings

\[
\text{rd}[x] = \text{rd}[x] \cup \text{tid}()
\]

Wrong dependence

\[
\text{Write}[V[x]] \rightarrow \text{tid}()
\]

\[
\text{rd}[x] \rightarrow \text{tid}()
\]
4 - Write-after-read Dependences: Handling Self-loop

\[
\text{rd}(x) = \text{rd}(x) \cup \text{tid}()
\]

\[
\text{V}(x) = \text{tid}()
\]

\[
\text{Write}[x]
\]

\[
\text{Read}[x]
\]

\[
\text{Read}[\text{V}(x)]
\]

\[
\text{rd}(x) \rightarrow \text{tid}()
\]

\[
\text{write-after-read dependence}
\]

\[
\text{read-after-write dependence}
\]
5 - Relaxed Memory Model

- RAW dependence tracing is barrier-free for x86-TSO
- fences required for weaker memory models
5 - Relaxed Memory Mode: Memory Ordering on x86-TSO

- Barrier-free for read-after-write dependences
- MFENCE for write-after-read dependences
RWTrace: Optimistic Shared Memory Dependence Tracing

- Technical highlights
  - precise WAW, RAW, WAR dependences tracing
  - wait-free thread-local read fast path
  - barrier-free on x86-TSO
  - scalable program instrumentation

- Implementation
  - upon LLVM, open source¹
  - as our experimental platform (like Octet)

¹http://github.com/jiangyy/rwtrace
Evaluation Results¹: Overhead

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Unmodified</th>
<th>RWTrace (RAW)</th>
<th>RWTrace (RAW+WAR)</th>
<th>MutexLock (LEAP)</th>
</tr>
</thead>
<tbody>
<tr>
<td>aget</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>pfscan</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>pbzip2</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>ocean</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>water</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>fft</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>radix</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>fluid</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>qsort</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>x264</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>knot</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>apache</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

¹ All experiments conducted on a 4×6-core Xeon machine.
Evaluation Results: Scalability

[Graph showing normalized running time vs. number of threads for unmodified, RWTrace (RAW), and RWTrace (RAW+WAR) for various benchmarks: ocean, water, fft, radix, fluid, qsort, x264.]
Shared Memory Dependences:
In Pursuit of Determinism

- Dynamic analysis of concurrent systems
- Deterministic replay, data race / atomicity violation / false sharing detection, etc.
- Shared memory dependence reduction
- Theoretical aspects
- Non-determinism control
- Software transactional memory
- Deterministic multi-threading
Thank You!