Hardware-assisted Tracing for Low Overhead Data Race Detection

Farzam Dorostkar with Pr. Michel Dagenais
May 16th 2022

Polytechnique Montreal
DORSAL Laboratory
Project Introduction

**Research Topic:** Low overhead memory bug detection using hardware tracing

**Current Track:**
Detecting data races in C/C++ programs that use POSIX pthreads
- Post-mortem data race detection using Intel Processor Trace (Intel PT)
Agenda

➢ Introduction
  • Data Race
  • Motivation
  • Intel Processor Trace (Intel PT)

➢ Methodology
  • Opportunities & Limitations
  • Algorithm
  • Hybrid Tracing
  • Preliminary Results
  • Tools

➢ Conclusion & Future work
Introduction
Introduction: Data Race

In multithread Programming:

- Shared variables allow threads to communicate quickly
- A bug when two+ threads access the same shared variable concurrently and at least one access is a write (Data Race!)
### Introduction: Data Race

#### In multithread Programming:

- Shared variables allow threads to communicate quickly.
- A bug when two+ threads access the same shared variable concurrently and at least one access is a write (Data Race!)

```c
int count = 0; //Shared variable

void *routine_one(void *arg) {
    count++;
}

void *routine_two(void *arg) {
    count++;
}

int main() {
    pthread_t t1, t2;
    pthread_create(&t1, NULL, &routine_one, NULL);
    pthread_create(&t2, NULL, &routine_two, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    return 0;
}
```

<table>
<thead>
<tr>
<th></th>
<th>t1 R</th>
<th>t2 R</th>
<th>t1 R</th>
<th>t1 W</th>
<th>t2 W</th>
<th>t2 R</th>
<th>t2 W</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Count = 1
Introduction: Motivation

Why a new data race detector?

❖ Current standard tools impose considerable overhead!
  ▪ ThreadSanitizer (TSan):
    • Slowdown: 5×-15× & Memory overhead: 5×-10×
  ▪ Helgrind:
    • Slowdown: 100× & Memory overhead: 20×

❖ Not usable in production + difficult to test applications under real-world loads

**Reason:** Sole reliance on heavy code instrumentation!

**Research question:** Can we reduce the need for code instrumentation in a data race detector by making use of hardware-assisted tracing?
Introduction: Intel Processor Trace (Intel PT)

- A hardware feature that logs information about software execution with minimal impact
- A non-intrusive means to trace control flow by generating a variety of packets
  - <5% performance overhead
- Decoder reconstructs the precise execution flow by combining PT packets with the binaries of the traced program
- Trace data is generated only for non-statically-known control flow changes
- Can store both cycle count and timestamp information
- No need to modify source code!
  - Run under Intel PT-enabled debug and profiling tools (like Linux perf & GDB)
Introduction: Intel Processor Trace (Intel PT)

Control Flow Tracing
- TNT (Taken Not-Taken): direct conditional branches (generates only a 1-bit indication)
- TIP (Target IP): target address of indirect branches, exception, and interrupts

<table>
<thead>
<tr>
<th>Instructions</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>push</td>
<td></td>
</tr>
<tr>
<td>mov</td>
<td></td>
</tr>
<tr>
<td>cmp</td>
<td></td>
</tr>
<tr>
<td>je .L1</td>
<td>TNT (Taken Not Taken)</td>
</tr>
<tr>
<td>mov</td>
<td></td>
</tr>
<tr>
<td>add</td>
<td></td>
</tr>
<tr>
<td>L1:</td>
<td></td>
</tr>
<tr>
<td>Call (edx) // virtual function</td>
<td>TIP (Target IP)</td>
</tr>
</tbody>
</table>
Reconstructing the Control Flow

Decoder can determine the exact execution flow from trace log

<table>
<thead>
<tr>
<th>Instructions</th>
<th>PT trace</th>
<th>Reconstructed Control Flow</th>
</tr>
</thead>
<tbody>
<tr>
<td>push</td>
<td>T</td>
<td>push</td>
</tr>
<tr>
<td>mov</td>
<td></td>
<td>mov</td>
</tr>
<tr>
<td>cmp</td>
<td>0x4012a4</td>
<td>cmp</td>
</tr>
<tr>
<td>je .L1</td>
<td></td>
<td>je .L1</td>
</tr>
<tr>
<td>mov</td>
<td></td>
<td>mov</td>
</tr>
<tr>
<td>add</td>
<td></td>
<td>add</td>
</tr>
<tr>
<td>L1: Call (edx) // virtual function</td>
<td></td>
<td>L1: Call (0x4012a4)</td>
</tr>
</tbody>
</table>
Methodology
Methodology: Opportunities & Limitations

What are the opportunities and limitations of using HW tracing for detecting data races?

What information is required for detecting data races?

How data races are detected?

Data race detection algorithms

• Happens-before
• Lockset
Methodology: Algorithm

Happens-before Algorithm

• Determines partial ordering between program events
• Inspired by Lamport’s Happens-before relation [1]
• For two events \( a \) and \( b \):
  i. If \( a \) and \( b \) in the same thread & \( a \) comes before \( b \) : \( a \rightarrow b \)
  ii. If \( (a, b) \) is a synchronization-pair (like lock/unlock the same mutex) : \( a \rightarrow b \)
  iii. If \( a \rightarrow b \) & \( b \rightarrow c \) : \( a \rightarrow c \) (Transitivity)

If \( a!\rightarrow b \) & \( b!\rightarrow a \) : \( a \) and \( b \) are concurrent!

• Used by many tools including TSan, Helgrind, and GO’s built-in data race detector

How data races are detected? 🔄

Methodology: Algorithm

Events of interest:

❖ Memory accesses:
  • reads and writes

❖ Synchronization:
  • Unlocking and locking the same mutex
  • Signaling a condition and waiting on the same condition
  • Broadcasting a condition and waiting on the same condition

Other required information:

❖ Control flow related information (like order of events)

What information is required for detecting data races?
Methodology: Hybrid Tracing

An Example Decoded PT Trace

Two threads updating the global object "count" while holding the mutual lock "mutex"

<table>
<thead>
<tr>
<th>TID</th>
<th>IP</th>
<th>Addr</th>
<th>Sym+off</th>
<th>Insn</th>
</tr>
</thead>
<tbody>
<tr>
<td>20295</td>
<td>0</td>
<td>4011dc</td>
<td>routine_two+0xc</td>
<td>mov $0x404060, %rdi</td>
</tr>
<tr>
<td>20295</td>
<td>401080</td>
<td>4011e6</td>
<td>routine_two+0x16</td>
<td>callq 0xfffffffffffffe9a</td>
</tr>
<tr>
<td>20295</td>
<td>0</td>
<td>4011eb</td>
<td>routine_two+0x1b</td>
<td>movl 0x40405c, %ecx</td>
</tr>
<tr>
<td>20295</td>
<td>0</td>
<td>4011f5</td>
<td>routine_two+0x25</td>
<td>movl %ecx, 0x40405c</td>
</tr>
<tr>
<td>20295</td>
<td>0</td>
<td>4011fc</td>
<td>routine_two+0x2c</td>
<td>mov $0x404060, %rdi</td>
</tr>
<tr>
<td>20295</td>
<td>401050</td>
<td>401209</td>
<td>routine_two+0x39</td>
<td>callq 0xfffffffffffffe47</td>
</tr>
<tr>
<td>20294</td>
<td>0</td>
<td>40118c</td>
<td>routine_one+0xc</td>
<td>mov $0x404060, %rdi</td>
</tr>
<tr>
<td>20294</td>
<td>401080</td>
<td>401196</td>
<td>routine_one+0x16</td>
<td>callq 0xfffffffffffffeea</td>
</tr>
<tr>
<td>20294</td>
<td>0</td>
<td>40119b</td>
<td>routine_one+0x1b</td>
<td>movl 0x40405c, %ecx</td>
</tr>
<tr>
<td>20294</td>
<td>0</td>
<td>4011a5</td>
<td>routine_one+0x25</td>
<td>movl %ecx, 0x40405c</td>
</tr>
<tr>
<td>20294</td>
<td>0</td>
<td>4011ac</td>
<td>routine_one+0x2c</td>
<td>mov $0x404060, %rdi</td>
</tr>
<tr>
<td>20294</td>
<td>401050</td>
<td>4011b9</td>
<td>routine_one+0x39</td>
<td>callq 0xfffffffffffffe97</td>
</tr>
</tbody>
</table>

Symbol | Addr
---|-------
count | 0x40405c
mutex | 0x404060
pthread_mutex_lock | 0x401080
pthread_mutex_unlock | 0x401050

- Control flow information
- Calls to synchronization functions
- Accesses to the global variable

No instrumentation required for this example
Methodology: Hybrid Tracing

In General:

- **Included in PT Trace:**
  - Control flow related information
  - Calls to synchronization functions of pthreads
  - Accesses to global objects

- **Not Included in PT Trace:**
  - Accesses to heap objects
  - Accesses to stack objects
  - Indirect pointer-based memory accesses

What are the opportunities and limitations of using HW tracing for detecting data races?
Methodology: Hybrid Tracing

Instrumenting Most Memory Accesses

At the current stage:

- Static instrumentation
- Using the LLVM infrastructure
- Instrumenting `load` and `store` instructions at IR level with dummy function calls

When the execution is over, and possibly on a different machine, the Intel PT trace data is decoded and analysed for possible data races.
Methodology: Preliminary Results

- Five C micro benchmarks
- No false positive or false negative report
- Proposed approach:
  - Noticeably less memory overhead, for all the five benchmarks
  - The least runtime overhead for four benchmarks
  - Performing the post-mortem analysis took between 3 minutes (for benchmark #4 / 11 MB) to 16 minutes (for benchmark #5 / 347 MB)
Conclusion & Future Work
Conclusion & Future Work

Conclusion:
- A hybrid data race detection tool that benefits from hardware-assisted tracing to minimize the need for code instrumentation
- Only instrumenting memory accesses. No need to instrument synchronization events.
- Promising preliminary results in comparison with Helgrind and TSan

Future Work:
- More efficient instrumentation + Dynamic Instrumentation
- Exploit the potential of static code analysis to specify memory accesses that do not affect the correctness of data race detection if not instrumented
- Investigating the potential of the PTWRITE instruction to further reduce the need for instrumentation
Questions?

farzam.dorostkar@polymtl.ca
https://github.com/FarzamDorostkar