ThreadMonitor: Low-overhead Data Race Detection using Intel PT

Farzam Dorostkar

September 17th 2023

Polytechnique Montreal

DORSAL Laboratory

About DORSAL

Research Areas

- Monitoring and Debugging of High Performance Distributed Heterogeneous Systems
- GPU Tracing and Profiling
- Scalable Trace Analysis and Visualization
- Low-overhead Runtime Verification
- Machine Learning-Powered Trace Analysis

Project Introduction

ThreadMonitor (TMon)

Post-mortem data race detector for C/C++ programs that use pthreads

- Traces the required runtime information for data race detection using Intel Processor Trace (Intel PT)
- Uses the trace data to emulate the same runtime verification performed by ThreadSanitizer (TSan)
- No direct impact on application memory usage
- Very low runtime overhead

Agenda

- Introduction
  - What Is a Data Race?
- Motivation
  - Need for Automated Data Race Detection
  - Why a New Tool?
- Methodology
  - Main Idea
  - Intel Processor Trace (PT)
  - Architecture
- Implementation
  - Compile-time Instrumentation of User Code
  - Intercepting Specific Library Functions
  - Postmortem Analyzer
- Latest Results
- Conclusion & Future Work

Introduction

Introduction: What Is a Data Race?

In Multithreaded Programs

- Threads typically share access to the application memory
- Shared memory enables efficient thread communication
- But also exposes a multithreaded program to *data races*
  1. Two threads access the same memory location without a timing constraint
     - Synchronization
     - Mutual exclusion
  2. At least one of the accesses is a write operation

Introduction: What Is a Data Race?

Example

- Two threads write to the shared variable `Global`
- No timing constraint ordering them, therefore a data race
- A concurrency error unless the resulting non-determinism is a design choice

```
int Global;

void *Thread1(void *x) {
 Global = 42;
 return x;
}

int main() {
 pthread_t t;
 pthread_create(&t, NULL,
Thread1, NULL);
 Global = 43;
 pthread_join(t, NULL);
 return 0;
}
```

tiny_race.c [1]

[1] https://clang.llvm.org/docs/ThreadSanitizer.html

Motivation

Motivation: Need for Automated Data Race Detection

Detecting data races can be extremely challenging for programmers

1. Ensure each access to shared memory follows proper timing constraints
    - Significantly complicated, even for a relatively simple project
2. Only particular thread interleavings may lead to the corruption of shared data
    - Possible to miss a data race even with comprehensive testing
    - A corrupted shared variable may not result in immediate failure

Motivation: Why a New Tool?

State-of-the-art tools cause considerable runtime and memory overhead!

- ThreadSanitizer (TSan)
  - Slowdown: **5×-15×** & Memory overhead: **5×-10×**
- Helgrind
  - Slowdown: **100×** & Memory overhead: **20×**

Cannot be used in many real-world testing scenarios!

- Some of our industrial partners cannot afford such overheads

Methodology

Methodology: Main Idea

ThreadMonitor (TMon)

A data race detector capable of performing the same analysis as TSan, but with very low-overhead

A postmortem tool

- Traces a program execution using Intel PT (ptwrite packets)
  - Very low overhead
- Processes the trace data to determine whether the traced execution exhibited data races
  - No data race detection analysis at runtime

No direct impact on application memory usage, very low runtime overhead

Methodology: Intel Processor Trace (PT)

A hardware feature that logs information about software execution with minimal impact

Facilities used by TMon:

1. PTWRITE (PTW) packet
    - **User-generated** 64-bit payload
    - `PTWRITE r64/m64` instruction
        - Sends the value of the operand passed to it to PT hardware to be encoded in a PTW packet
        - Previously introduced in Atom, now available in Alder Lake (12th generation)
    - Very low overhead (only 2 CPU cycles on our machine)
2. Metadata
    - Thread/Process IDs

Methodology: Architecture

What to Trace?

The same program events tracked by TSan

- The same runtime information captured by TSan for each event

How to Trace?

Instrument each event of interest with a `ptwrite` instruction

- Most significant byte of its payload indicates the event type (less than 255 event types)
- Remaining seven bytes store the required runtime information to analyze that event

Three Main Components of TMon:

1. Compile-time Instrumentation of User Code
    - Instrumenting memory accesses in user code (similar to TSan)
2. Intercepting Specific Library Functions
    - Library functions related to imposing timing constraints among threads, or accessing memory (similar to TSan)
3. Postmortem Analyzer
    - Analyzes the trace data

Implementation

Implementation: Compile-time Instrumentation of User Code

Compile-time instrumentation at LLVM IR level

- Function pass
- Identify and instrument various types of memory accesses within user code
- Instrumenting function entry and exit points if necessary

Main parts:

1. Assessing Instrumentation Eligibility of a Function
2. Function Traversal
3. Instrumenting Non-atomic Memory Accesses
4. Instrumenting Atomic Memory Operations
5. Instrumenting Function Entry and Exit

Implementation: Compile-time Instrumentation of User Code

1. Assessing Instrumentation Eligibility of a Function

Subject to instrumentation exemption if the function possesses either of the following attributes:

- `Naked`
  - Indicates the absence of standard prologue and epilogue sequences
  - Unable to instrument function entry and exit points
  - Similar to TSan
- `DisableTMonInstrumentation`
  - Designed to provide programmers with the flexibility to selectively leave certain functions uninstrumented
  - TSan provides a similar function attribute

Implementation: Compile-time Instrumentation of User Code

2. Function Traversal

Once qualified for instrumentation, the pass traverses the function to identify the instructions engaged in accessing memory.

- **TMon targets the same set of instructions as TSan**
  - Non-atomic memory accesses
  - Atomic memory operations
- TSan detects three redundancy cases in non-atomic accesses
  1. Read-before-write happening within the same basic block, with no calls occurring between them
     - The read instruction can be safely excluded from instrumentation
     - The write instruction is marked as a compound access
  2. Reading an address that points to constant data
  3. Access addressable variables that are not captured
     - Such variables cannot be referenced from a different thread
- TMon employs the same redundancy analysis, thereby **instruments exactly the same instructions as TSan**

Implementation: Compile-time Instrumentation of User Code

3. Instrumenting Non-atomic Memory Accesses

TSan inserts a call to a specialized runtime library function immediately before the access occurs.

- The data race detection logic requires to obtain six properties pertaining to each non-atomic access
  1. Access type (read or write)
  2. Access size (supports access sizes of 1, 2, 4, 8, and 16 bytes)
  3. Whether aligned
  4. Whether a compound access
  5. Whether accesses a volatile memory location
  6. Accessed address
- The first five properties contribute to a total of **50** distinct types of non-atomic accesses.
- TSan encodes these five properties by employing a dedicated instrumentation function for each specific case.
  - `__tsan_read1()` is used to instrument non-volatile read operations of size one byte
- The last property (accessed address) is passed to the corresponding instrumentation function.

Implementation: Compile-time Instrumentation of User Code

3. Instrumenting Non-atomic Memory Accesses (Cont.)

TMon inserts a single `ptwrite` instruction immediately before the access occurs.

- Supports the same 50 different types of non-atomic memory accesses
- Traces the same six properties for each access
  - The most significant byte of the payload cumulatively encodes the first five properties
    - Allocating 50 unique values
    - Each exclusively associated with one of the 50 instrumentation functions employed by TSan
  - The six least significant bytes of the payload store the accessed address
- Enabling its postmortem analyzer to apply the same data race detection logic implemented in the TSan runtime for analyzing non-atomic accesses

Implementation: Compile-time Instrumentation of User Code

4. Instrumenting Atomic Memory Operations

TSan inserts a call to a specialized runtime library function immediately preceding the occurrence of the atomic operation.

- The data race detection logic requires to obtain three properties pertaining to each atomic operation
  1. Operation type (atomic load, atomic store, atomic read-modify-write (RMW), and atomic compare-and-swap (CAS))
  2. Access size (supports access sizes of 1, 2, 4, 8, and 16 bytes)
  3. Accessed address
- The first two properties contribute to a total of **20** distinct types of atomic operations.
- Encodes these two properties by employing a dedicated instrumentation function for each specific case.
  - `__tsan_atomic8_load()` is used to instrument atomic load operations of size 8 bits
- The last property (accessed address) is passed to the corresponding instrumentation function.

Implementation: Compile-time Instrumentation of User Code

4. Instrumenting Atomic Memory Operations (Cont.)

TMon inserts a single `ptwrite` instruction immediately before the atomic operation occurs.

- Supports the same 20 different types of atomic operations
- Traces the same three properties for each operation
  - The most significant byte of the payload cumulatively encodes the first two properties
    - Allocating 20 unique values
  - The six least significant bytes of the payload store the accessed address
- Enabling its postmortem analyzer to apply the same data race detection logic implemented in the TSan runtime for analyzing atomic operations

Implementation: Compile-time Instrumentation of User Code

5. Instrumenting Function Entry and Exit

TSan instruments entry and exit points of a function if it contains instrumented memory accesses.

- Preserve a precise stack trace for every access (used in data race reports)
- Entry: inserts a call to `__tsan_func_entry()` with the return address of the current function passed to it
- Exit: marks a function exit by invoking `__tsan_func_exit()`

TMon follows the same behavior.

- Entry: inserts a single `ptwrite` instruction
  - The most significant byte of the payload serves as an indicator for a function entry event
  - The six least significant bytes store the return address of the current function
- Exit: inserts a single `ptwrite` instruction
  - The most significant byte of the payload serves as an indicator for a function entry event

Implementation: Compile-time Instrumentation of User Code

Example: TMon vs. TSan

Inline assembly, not a function call!

```
define dso_local ptr @Thread1(ptr noundef %x) {
entry:
%0 = call ptr @llvm.returnaddress(i32 0)
%1 = ptrtoint ptr %0 to i64
%ptw.funcentry = or i64 %1, 72057594037927936
call void asm "ptwriteq $0", "rm"(i64 %ptw.funcentry)
...
call void asm "ptwriteq $0", "rm"(i64 or (i64 ptrtoint (ptr @Global to i64), i64 8646911284551352320))
store i32 42, ptr @Global, align 4
...
call void asm "ptwriteq $0", "rm"(i64 144115188075855872)
ret ...
}

define dso_local i32 @main() {
entry:
%0 = call ptr @llvm.returnaddress(i32 0)
%1 = ptrtoint ptr %0 to i64
%ptw.funcentry = or i64 %1, 72057594037927936
call void asm "ptwriteq $0", "rm"(i64 %ptw.funcentry)
...
%t = alloca i64, align 8
...
call void asm "ptwriteq $0", "rm"(i64 or (i64 ptrtoint (ptr @Global to i64), i64 8646911284551352320))
store i32 43, ptr @Global, align 4
%2 = ptrtoint ptr %t to i64
%ptw.rw = or i64 %2, 576460752303423488
call void asm "ptwriteq $0", "rm"(i64 %ptw.rw)
%3 = load i64, ptr %t, align 8
...
call void asm "ptwriteq $0", "rm"(i64 or (i64 ptrtoint (ptr @Global to i64), i64 5044031582654955520))
%4 = load i32, ptr @Global, align 4
call void asm "ptwriteq $0", "rm"(i64 144115188075855872)
ret ...
}
```

tiny_race_tmon.ll

```
define dso_local ptr @Thread1(ptr noundef %x) {
entry:
%0 = call ptr @llvm.returnaddress(i32 0)
call void @__tsan_func_entry(ptr %0)
...
call void @__tsan_write4(ptr @Global)
store i32 42, ptr @Global, align 4
...
call void @__tsan_func_exit()
ret ...
}

define dso_local i32 @main() {
entry:
%0 = call ptr @llvm.returnaddress(i32 0)
call void @__tsan_func_entry(ptr %0)
...
%t = alloca i64, align 8
...
call void @__tsan_write4(ptr @Global)
store i32 43, ptr @Global, align 4
call void @__tsan_read8(ptr %t)
%1 = load i64, ptr %t, align 8
...
call void @__tsan_read4(ptr @Global)
%2 = load i32, ptr @Global, align 4
call void @__tsan_func_exit(),
ret ...
}
```

tiny_race_tsan.ll

Implementation: Intercepting Specific Library Functions

TSan intercepts common library functions that impose a timing constraint or access memory.

- Most importantly `pthread` functions
- Highly integrated with the internal race detection logic
- Defined as a static/shared library (depending on the compiler)

```
int __interceptor_function(...) {

 // Call the actual function.
 res = REAL(function)(...);

 // Update the status of the race detection logic.
 ...

 return res;
}
```

Implementation: Intercepting Specific Library Functions

TMon employs interceptors as well.

- No race detection analysis at runtime despite TSan interceptors
- **Meant to record required runtime information using a `ptwrite` packet**
  - Depends on the function being intercepted
  - Generally: some attribute passed to it and the return value
- Symbol interposition to redirect such function calls to its own implementation
- `__tmon_interceptor_function` has a weak alias of the same name as the intercepted function
- Defined as a static library

```
void tmon_interceptor_function() {

 // Call the actual function.
 res = REAL(function)(...);

// Record the required runtime information.
 asm volatile ("ptwrite %0"...);
}
```

Implementation: Postmortem Analyzer

Processes the trace data to determine whether the program execution exhibited data races.

- Reconstructs the sequence of program events
  - Using the information encoded within the `ptwrite` packets and the associated metadata

Builds upon the data race detection logic used by TSan (reuses parts of TSan RTL)

- Happens-before based algorithm
- Based on the happened-before relation proposed by Lamport

Enhancing its coverage through the introduction of novel algorithmic contributions

Implementation: Postmortem Analyzer

Mitigating Data Race Loss in TSan

TSan uses *shadow cells* to keep track of memory accesses.

- Every consecutive eight bytes of application memory are mapped to four shadow cells
- Each shadow cell encodes an access to the associated application memory region
- Upon detecting a new memory access, it is compared with prior conflicting accesses encoded by shadow cells
- A notable factor contributing to data race loss in TSan is the necessity to overwrite shadow cells due to their limited quantity
- TSan uses a random selection strategy to overwrite shadow cells

TMon employs a postmortem adaptation of the shadow cell paradigm, but proposes a refined approach.

- Allocating More Shadow Cells
    - Reduces the need to overwrite shadow cells
- Better Overwriting Policy
    - Selecting the shadow cell associated with the access involving the least number of bytes
        - Reduces the risk of overlapping with subsequent accesses

Latest Results

Latest Results

Fourier Transform [1]

- Different number of threads, different number of input values

Benchmark	#Threads	#Input Values	Execution Time (sec)			Memory Overhead		Post-mortem Overhead
			Native	TMon*	TSan	TMon**	TSan	TMon***
Fourier Transform	5	2^{15}	8.0	9.1	38.1	0%	4.7×	52%
		2^{16}	31.9	36.3	152.0	0%	4.8×	65%
	10	2^{15}	5.4	6.3	67.1	0%	5.1×	53%
		2^{16}	21.5	23.3	281.4	0%	5.5×	68%
	15	2^{15}	3.8	4.3	63.7	0%	6.0×	60%
		2^{16}	15.2	18.4	263.5	0%	6.2×	72%
Avr. Overhead				**1.15×**	**11.4×**	**-**	**5.4×**	

* Includes the overhead of collecting traces using `perf`

** No direct impact on the application, but there are Intel PT buffers for collecting the trace

*** In comparison to the native execution time

[1] https://github.com/EstellaPaula/FFT-parallelized-with-PThread-API

Conclusion & Future Work

Conclusion & Future Work

Conclusion

- TMon: A low-overhead postmortem data race detector for C/C++ programs
- Based on low-overhead ptwrite instrumentation
- Encoding the required runtime information for data race detection as ptwrite payloads
- Much less runtime and memory overhead compared to TSan

Future Work

- A similar approach may be adapted to design post-mortem tools that emulate other runtime verification tools, such as AddressSanitizer (ASan)

Thanks!

Questions? Comments?

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