Low-overhead Memory Error Detection using Intel PT: From ThreadMonitor to AddressMonitor

Farzam Dorostkar with Prof. Michel Dagenais

Dec. 7th 2023

Polytechnique Montreal

DORSAL Laboratory

Recap

ThreadMonitor (TMon)

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

- Offers the same data race detection analysis as ThreadSanitizer (TSan) but with significantly lower overhead
- Traces the same runtime information captured by TSan for analysis purposes
  - Uses Intel's `ptwrite` packets
  - User-generated 64-bit payload
- Uses the trace data to emulate the same runtime verification performed by TSan
- No direct data memory overhead, minimal instruction memory overhead, very low runtime overhead

Since Last Meeting

Previous Track: ThreadMonitor (TMon)

- Completed remaining implementation details
- Conducted a more comprehensive evaluation study
  - Compared its performance with TSan on a set of SPEC CPU 2017 benchmarks
  - Explored runtime effects introduced by the tracer (Linux perf)
- Presented it at Tracing Summit 2023
- Documented it as a paper

Current Track: AddressMonitor (AMon)

- Recently started, initiating the groundwork
- Goal: detecting some other common memory errors

TMon Updates: Compile-time Instrumentation

TMon Updates: Compile-time Instrumentation

Compile-time instrumentation at LLVM IR level

- Function pass
- Identify and instrument various types of memory accesses within user code

Main parts:

- Assessing Instrumentation Eligibility of a Function
- **Function Traversal**
- **Instrumenting Non-atomic Memory Accesses**
- Instrumenting Atomic Memory Operations
- Instrumenting Function Entry and Exit

TMon Updates: Compile-time Instrumentation

Function Traversal

The pass traverses the function to identify the memory access instructions.

- **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**

TMon Updates: Compile-time Instrumentation

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_read4()` is used to instrument non-volatile read operations of size four bytes
- The last property (accessed address) is passed to the corresponding instrumentation function.

TMon Updates: Compile-time Instrumentation

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

TMon Updates:
Enhanced Detection Coverage

TMon Updates: Enhanced Detection Coverage

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
- Overwriting shadow cells is a notable factor contributing to missing data races in TSan
- TSan uses a random selection strategy to overwrite shadow cells

TMon Updates: Enhanced Detection Coverage

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
  - The idea is also applicable to TSan

TMon Updates:
New Evaluation Study

TMon Updates: New Evaluation Study

- A set of four SPEC CPU 2017 benchmarks
- Executable size, execution time, and memory consumption
- Native compilation, under Tsan, and under TMon

Benchmark	Native			TSan			TMon		
	Exec. Size	Time (sec)	MRSS (MB)	Exec. Size (x)	Time (x)	MRSS (x)	Exec. Size (x)	Time (x)	MRSS (x)
mcf	113 KB	5.9	292	13.3 x	3.7 x	2.9 x	1.2 x	2.8 x	2.1 x
lbm	50 KB	1.0	420	28.5 x	4.1 x	3.0 x	1.9 x	5.1 x	1.4 x
namd	3 MB	1.8	160	1.8 x	7.5 x	3.1 x	1.2 x	2.2 x	1.5 x
parest	75 MB	2.0	99	1.3 x	9.0 x	4.7 x	1.1 x	2.9 x	1.5 x
Average				11.2 x	6.1 x	3.5 x	1.3 x	3.2 x	1.6 x

AddressMonitor (AMon)

AddressMonitor (AMon)

Post-mortem memory error detector for C/C++ programs

- Initial focus on detecting out-of-bound accesses
- Adaptation of a similar approach to TMon
  - Traces the required runtime information for memory error detection using Intel PT
  - Uses the trace data to emulate the same runtime verification performed by AddressSanitizer (ASan)
    - ❑ Motivation: ASan cause considerable memory and runtime overhead
- Does not require shadow memory
- Expected to have minimal instruction memory overhead
- Expected to have very low runtime overhead

AddressMonitor (AMon)

Two main components:

1. Compile-time instrumentation of user code
   - At LLVM IR level (function pass)
   - Instruments the same type of memory accesses monitored by ASan
   - Uses a `ptwrite` packet to record the required runtime information for each access

2. Postmortem analyzer
   - Performs the same analysis as Asan

Thanks!

Questions? Comments?

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