ThreadMonitor: Low-overhead Data Race Detection using Intel PT

Farzam Dorostkar with Pr. Michel Dagenais

Jun 1st 2023

Polytechnique Montreal

DORSAL Laboratory

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
  - Motivation
  - Intel Processor Trace (Intel PT)
- Methodology
  - What to Trace?
  - User Code
  - Standard Library Functions
  - Latest Results
- Conclusion & Future work

Introduction

Introduction: Motivation

Why a new data race detector?

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×

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

Reason: Costly software instrumentation

Introduction: Intel Processor Trace

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

Provides different facilities:

1. A non-intrusive means to trace the exact control flow
   - Trace data is generated only for non-statically-known control flow changes
2. 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)
3. Metadata
   - Thread/Process IDs, pc

Used by TMon

Methodology

Methodology: What to trace?

Two types of events:

1. Memory accesses
   - Read/write, atomic/non-atomic, accessed addresses, etc.
2. Synchronization
   - Thread creation/join, mutual lock, etc.

Two domains to monitor: user code & library functions

- Each domain is monitored differently
- We will compare TMon vs. TSan

Methodology: User Code

TSan:

Compile-time instrumentation of events of interest at IR level

- Different types and sizes of memory accesses, function entries/exits, etc. (82 events in total)
- LLVM pass [1] to insert calls to TSan runtime library right before those events

```
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 Global;
}
```

tiny_race.c [2]

```
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

[1] llvm-project/llvm/lib/Transforms/Instrumentation/ThreadSanitizer.cpp
[2] https://clang.llvm.org/docs/ThreadSanitizer.html

Methodology: User Code

TSan:

At runtime, the instrumentations trigger the race detection logic and/or update the status of the analyzer.

Triggering the race detection logic

```
void __tsan_write4(void *addr) {
 MemoryAccess(cur_thread(), addr, 4, kAccessWrite);
}
```

tsan_interface.inc [1]

TSan RTL (race detection logic)

```
bool CheckRaces(ThreadState* thr, RawShadow* shadow_mem, Shadow cur, AccessType typ) {
 /*Compare current access against previous accesses*/
}

void MemoryAccess(ThreadState* thr, uptr addr, uptr size, AccessType typ) {
 RawShadow* shadow_mem = MemToShadow(addr); /*where prev. accesses are encoded*/
 …
 CheckRaces(thr, shadow_mem, cur, typ);
}
```

tsan_rtl_access.cpp [2]

```
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 ...
}
```

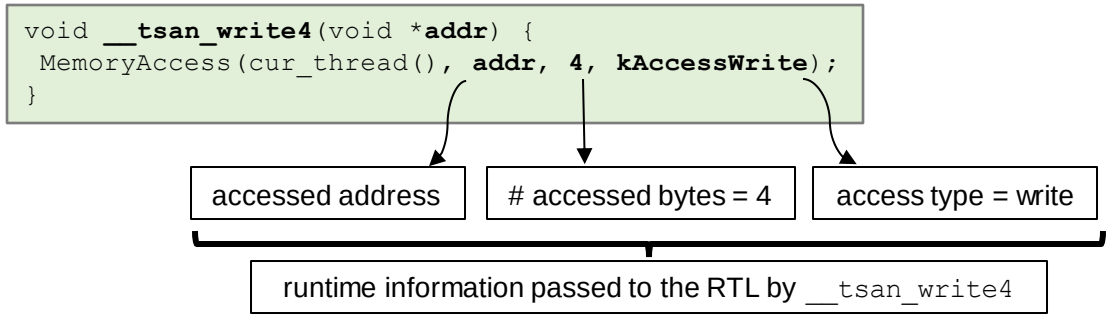
[1] llvm-project/compiler-rt/lib/tsan/rtl/tsan_interface.inc
[2] llvm-project/compiler-rt/lib/tsan/rtl/tsan_rtl_access.cpp

Methodology: User Code

TMon:

Compile-time instrumentation of the same 82 events as in TSan at IR level

- LLVM pass to insert a `ptwrite` instruction right before those events
- The **8-byte payload** encodes:
  1. Event type (most significant byte)
  2. Runtime information required to later analyze that event
- We use TSan RTL as part of our post-mortem analyzer



- `ptwrite` payload to instrument a write of size 4:

Event Type : 1 byte	(Unused) 1 byte	**Accessed Address : 6 bytes**

- Similar idea for read of size 4 and 8

Methodology: User Code

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 864691128455135232))
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 864691128455135232))
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 504403158265495552))
%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

Methodology: Standard Library Functions

TSan:

Intercepts library functions that impose synchronization or access memory

- `pthread_*` functions, memory allocation/deallocation, etc.
- Symbol interposition to redirect such function calls to its own implementation
- `__interceptor_name` has a weak alias `name` of the same name as the intercepted function

Example 1 (`pthread_join`):

NOT a data race!

```
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 Global;
}
```

Methodology: Standard Library Functions

TSan:

Intercepts library functions that impose synchronization or access memory

- `pthread_*` functions, memory allocation/deallocation, etc.
- Symbol interposition to redirect such function calls to its own implementation
- `__interceptor_name` has a weak alias `name` of the same name as the intercepted function

Example 2 (`pthread_create`):

NOT a data race!

```
int Global;

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

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

Methodology: Standard Library Functions

TMon:

TSan interceptors are more challenging to replicate using `ptwrite` packets

- Not just simple wrappers
- Highly integrated with the internal race detection logic
- Each interceptor needs its own special treatment

```
int __interceptor_name(...) {

 /*Prepare the analyzer to call the actual function*/

 res = REAL(name)(...);

 /*Update thread-local and/or global status*/

 return res;
}
```

Methodology: Standard Library Functions

TMon:

Example: Replicating the behavior of `__interceptor_pthread_create`

- How to Initialize the status of a new thread (contains information like initial vector clock)?
- An snapshot of the status of the parent when calling `pthread_create`
- TSan interceptor stores it in a global object (`ctx`), assigns an internal tid to it, and passes the tid to the new thread

```
int __interceptor_pthread_create(void *(*callback)(void*), void * param,...) {

 /*Prepare the analyzer to call the actual function*/
 ThreadParam p;          // passed to __tsan_thread_start_func
 p.callback = callback; // user thread function
 p.param = param;        // passed to the user thread function
 p.tid = InvalidTid;     // internal thread identifier allocated by ThreadRegistry

 res = REAL(pthread_create)(__tsan_thread_start_func, &p, ...);

 /*Update thread-local and/or global status*/
 if(res == 0) {
  p.tid = ThreadCreate(...);
  p.created.Post();
  p.started.Wait();
 }

 return res;
}
```

tsan_interceptors_posix.cpp [1]

```
void *__tsan_thread_start_func(void *arg) {

 ThreadParam *p = (ThreadParam*)arg;

 ...

 p->created.Wait();
 ThreadStart(p->tid, ...);
 // The initial status of the new thread is retrieved from ctx.
 p->started.Post();

 /*Calls the user function*/

 return ...;
}
```

tsan_interceptors_posix.cpp [1]

[1] llvm-project/compiler-rt/lib/tsan/rtl/tsan_interceptors_posix.cpp

Methodology: Standard Library Functions

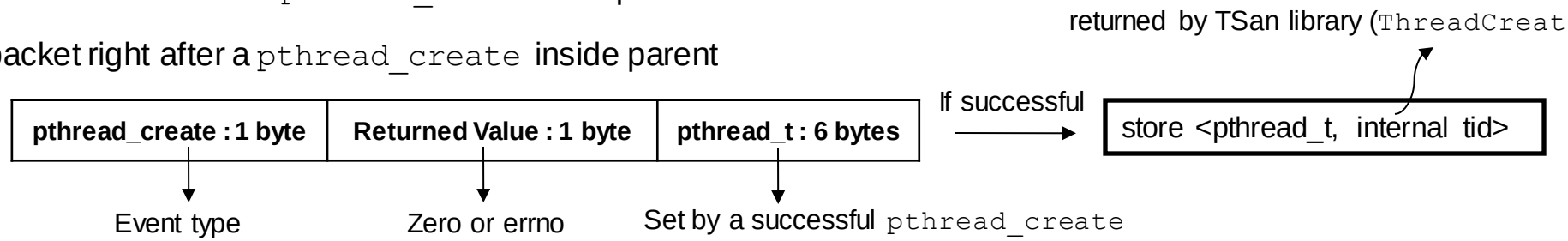
TMon:

Example: Replicating the behavior of `__interceptor_pthread_create`

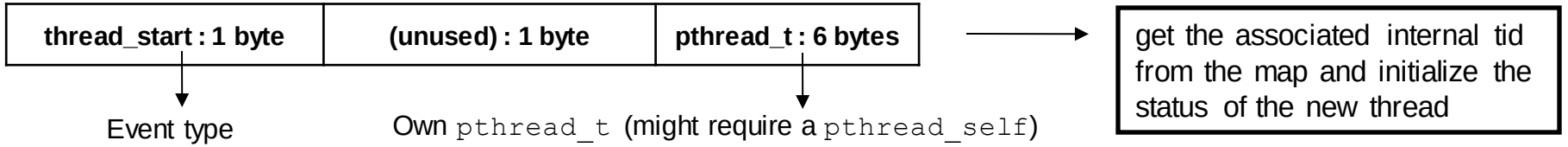
Proposed solution: associate the internal tid with the `pthread_t` handle

Two `ptwrite` packets and a call to `pthread_self` are required:

1. One packet right after a `pthread_create` inside parent



2. One packet at child thread startup



Methodology: Standard Library Functions

TMon:

Detecting calls to library functions of interest

1. Compile-time instrumentation of user code
    - Use an LLVM pass to find the calls and insert the required instrumentation
    - Indirect calls cannot be resolved at compile-time

```
%call = call i32 @pthread_create(ptr noundef %t, ptr noundef null, ptr noundef @Thread1, ptr noundef null)
```

```
%call = call i32 @pthread_create(ptr noundef %t, ptr noundef null, ptr noundef @Thread1, ptr noundef null)
%load.ptw = load i64, ptr %t, align 8
%ptw.pthread_create = or i64 %load.ptw, 5980780305148018688
call void asm "ptwriteq $0", "rm"(i64 %ptw.pthread_create)

...

define dso_local ptr @Thread1(ptr noundef %x) #0 !dbg !16 {
entry:
%ptw.ptid = call i64 @pthread_self(), !dbg !20
%ptw.ptid_payload = or i64 %ptw.ptid, 6052837899185946624, !dbg !20
call void asm "ptwriteq $0", "rm"(i64 %ptw.ptid_payload) #4, !dbg !20
```

Methodology: Standard Library Functions

TMon:

Detecting calls to library functions of interest

1. Compile-time instrumentation of user code
   - Use an LLVM pass to find the calls and insert the required instrumentation
   - Indirect calls cannot be resolved at compile-time
2. Simple function wrappers
   - One extra function call, but very flexible

```
void wrapper_name(...) {

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

 asm volatile ("ptwrite %0"...);

}
```

Methodology: Standard Library Functions

TMon:

Detecting calls to library functions of interest

1. Compile-time instrumentation of user code
   - Use an LLVM pass to find the calls and insert the required instrumentation
   - Indirect calls cannot be resolved at compile-time
2. Simple function wrappers
   - One extra function call, but very flexible
3. Instrumenting the library
   - Less flexible, but also less overhead

```
int __pthread_create_2_1(...) {

…

asm volatile ("ptwrite %0"...);

return
```

```
int start_thread(...) {

…

asm volatile ("ptwrite %0"...);
// Calls user function.

return
```

Methodology: 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:
  - A post-mortem data race detector based on low-overhead PTWRITE instrumentation
  - Encoding the required runtime information for data race detection as 64-bit payloads
  - Much less runtime and memory overhead compared to TSan
- Future Work:
  - Finish monitoring the remaining library functions
  - Report generation and opportunities for producing better reports than TSan

Questions?

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