Low Overhead Data Race Detection using Intel PT

Farzam Dorostkar with Pr. Michel Dagenais

Dec 8th 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
  - Motivation
  - Intel Processor Trace (Intel PT)
- Methodology
  - Recap and Current roadmap
  - Replicating TSan using PTWRITE
  - Accuracy-Overhead Trade-offs in TSan
  - Preliminary Results
- Conclusion & Future work

Introduction

Introduction: Motivation

Why a new data race detector?

State-of-the-art tools

1. Cause considerable runtime 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

Solution: Using low-overhead hardware tracing

2. Compromise accuracy for less overhead!
   - I could detect two such trade-offs in TSan

Reason: Race detection analysis happens at runtime

Solution: Post-mortem analysis enables a more comprehensive analysis

Introduction: Intel Processor Trace

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

1. A non-intrusive means to trace the exact control flow
   - Trace data is generated only for non-statically-known control flow changes
   - Generates a variety of packets
     - TNT (Taken Not-Taken) : direct conditional branches
     - TIP (Target IP) : target address of indirect branches, exception, and interrupts
   - <5% performance overhead
2. Metadata
   - Thread/Process IDs, Timing information (cycle count and timestamp), pc
3. 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)

Methodology

Methodology: Recap and Current Roadmap

In my **previous** presentation:

- Analysing the reconstructed execution flow at assembly level for data races
- Two main challenges:
  1. Analyzing assembly is difficult, especially for memory accesses
  2. Tracing execution flow produces high volumes of PT trace data

Current roadmap:

- PTWRITE was a game changer!
- Low-overhead PTWTITE instrumentation instead of tracing the execution flow
  1. Shift from analyzing executed instructions to analyzing 64-bit PTW packets
  2. Much less PT trace data is generated (only generating PTW packets)
- In a sense similar to what TSan does
  - Compile-time instrumentation + runtime library

Research question: Can we replicate TSan's functionality using PTWRITE instrumentation and a post-mortem analyser?

- Current TSan (v3) is very different from the first version and yet not documented!
- Long hours of reading the source code and debugging...

Methodology: Replicating TSan using PTWRITE

How does TSan check for data races?

1. Instrumenting memory accesses

```
#include <pthread.h>
#include <stdlib.h>

void* routine_child(void *count) {
++ *((int *)count);
return 0;
}

int main (void){
int *ptr = (int *)malloc(sizeof *ptr);
*ptr = 100;

pthread_t t;
pthread_create(&t, NULL, &routine_child, (void *)ptr);

pthread_join(t, NULL);
return 0;
}
```

```
<_Z13routine_childPv>:
                ...
call    <__tsan_write4@plt>
```

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

```
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);
}
```

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

Methodology: Replicating TSan using PTWRITE

How does TSan check for data races?

1. Instrumenting memory accesses

```
#include <pthread.h>
#include <stdlib.h>

void* routine_child(void *count) {
++ *((int *)count);
return 0;
}

int main (void){
int *ptr = (int *)malloc(sizeof *ptr);
*ptr = 100;

pthread_t t;
pthread_create(&t, NULL, &routine_child, (void *)ptr);

pthread_join(t, NULL);
return 0;
}
```

```
<_Z13routine_childPv>:
                ...
call    <__tsan_write4@plt>
```

Need to provide the same arguments passed to the logic

Triggering the race detection logic

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

Race detection logic (aiming to reuse in our post-mortem analysis)

```
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);
}
```

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

Methodology: Replicating TSan using PTWRITE

How does TSan check for data races?

1. Instrumenting memory accesses

```
#include <pthread.h>
#include <stdlib.h>

void* routine_child(void *count) {
++ *((int *)count);
return 0;
}

int main (void){
int *ptr = (int *)malloc(sizeof *ptr);
*ptr = 100;

pthread_t t;
pthread_create(&t, NULL, &routine_child, (void *)ptr);

pthread_join(t, NULL);
return 0;
}
```

```
<_Z13routine_childPv>:
                ...
call    <__tsan_write4@plt>
```

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

accessed address | # accessed bytes = 4 | access type = write

information provided by the instrumenting function, i.e. `__tsan_write4`

Methodology: Replicating TSan using PTWRITE

How does TSan check for data races?

1. Instrumenting memory accesses

```
#include <pthread.h>
#include <stdlib.h>

void* routine_child(void *count) {
++ *((int *)count);
return 0;
}

int main (void){
int *ptr = (int *)malloc(sizeof *ptr);
*ptr = 100;

pthread_t t;
pthread_create(&t, NULL, &routine_child, (void *)ptr);

pthread_join(t, NULL);
return 0;
}
```

```
<_Z13routine_childPv>:
                ...
call    <__tsan_write4@plt>
```

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

accessed address | # accessed bytes = 4 | access type = write

information provided by the instrumenting function, i.e. `__tsan_write4`

Proposed idea: replace the instrumenting function, i.e. `__tsan_write4`, with a PTWRITE instruction with the following payload:

Access Type (__tsan_write4) ID: 2 bytes	Accessed Address: 6 bytes

Methodology: Replicating TSan using PTWRITE

How does TSan check for data races?

1. Instrumenting memory accesses
2. Intercepting pthreads and memory allocation/deallocation functions
   - `__interceptor_pthread_create, __interceptor_malloc, ...`

```
<_Z13routine_childPv>:
                ...
call    <__tsan_write4@plt>
```

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

- TSan's internal object called `ThreadStatus`
- Thread local (2KB per thread)
- Stores information about current status of the thread, required by the race detection logic
  - like `VectorClock`
- TSan updates it by intercepting pthreads and memory allocation/deallocation functions

```
#include <pthread.h>
#include <stdlib.h>

void* routine_child(void *count) {
++ *((int *)count);
return 0;
}

int main (void){
int *ptr = (int *)malloc(sizeof *ptr);
*ptr = 100;

pthread_t t;
pthread_create(&t, NULL, &routine_child, (void *)ptr);

pthread_join(t, NULL);
return 0;
}
```

Methodology: Replicating TSan using PTWRITE

How does TSan check for data races?

1. Instrumenting memory accesses
2. Intercepting pthreads and memory allocation/deallocation functions
   - `__interceptor_pthread_create, __interceptor_malloc, ...`

```
<_Z13routine_childPv>:
                ...
call    <__tsan_write4@plt>
```

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

```
#include <pthread.h>
#include <stdlib.h>

void* routine_child(void *count) {
++ *((int *)count);
return 0;
}

int main (void){
int *ptr = (int *)malloc(sizeof *ptr);
*ptr = 100;

pthread_t t;
pthread_create(&t, NULL, &routine_child, (void *)ptr);

pthread_join(t, NULL);
return 0;
}
```

- TSan's internal object called `ThreadStatus`
- Thread local (2KB per thread)
- Stores information about current status of the thread, required by the race detection logic
  - like `VectorClock`
- TSan updates it by intercepting pthreads and memory allocation/deallocation functions

Proposed idea: replace each interceptor with a PTWRITE instruction with following payload and maintain the same object in the post-mortem decoder

pthread Function ID: 2 bytes	`pthread_t` Handler: 6 bytes

Memory Alloc/Dealloc Function ID: 2 bytes	Alloc/Deallocated Address: 6 bytes

Methodology: Accuracy-Overhead Trade-offs in TSan

Two main Trade-offs:

1. Algorithm-wise
    - The algorithm used to detect data races

2. Implementation-wise
    - How the race detection algorithm is implemented

Methodology: Accuracy-Overhead Trade-offs in TSan

1. Algorithm-wise

Data race detection algorithms:

- Happens-before
  - Inspired by Lamport's Happened-before partial order [1]
  - Checks if feasible to define a happened-before relation between two conflicting memory accesses
- Lockset
  - Relies on simple locking discipline
  - Checks if each shared memory location is consistently protected by at least one lock

TSan runtime library:

- Used to implement a hybrid algorithm
- Latest release (v3) is pure Happens-before
  - Less runtime overhead, but also less race detection coverage!

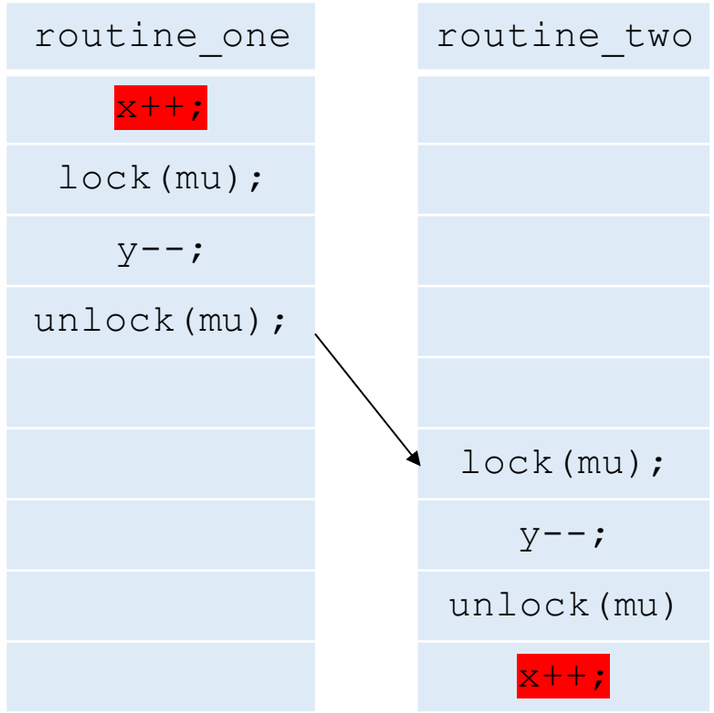
[1] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," Commun. ACM, 21(7): 558–565, 1978.

Methodology: Accuracy-Overhead Trade-offs in TSan

1. Algorithm-wise (Example)

Happens-before misses the potential race on accessing variable x, i.e. false negative!

```
#include <pthread.h>

pthread_mutex_t mu;

int x = 0, y = 0;

void* routine_one(void *arg) {
x++;
pthread_mutex_lock(&mu);
y--;
pthread_mutex_unlock(&mu);
return NULL;
}

void* routine_two(void *arg) {
pthread_mutex_lock(&mu);
y--;
pthread_mutex_unlock(&mu);
x++;
return NULL;
}

int main (void){
pthread_t t[2];
pthread_mutex_init(&mu, NULL);

pthread_create(&t[0], NULL, &routine_one, NULL);
pthread_create(&t[1], NULL, &routine_two, NULL);

pthread_join(t[0], NULL);
pthread_join(t[1], NULL);

pthread_mutex_destroy(&mu);
return 0;
}
```

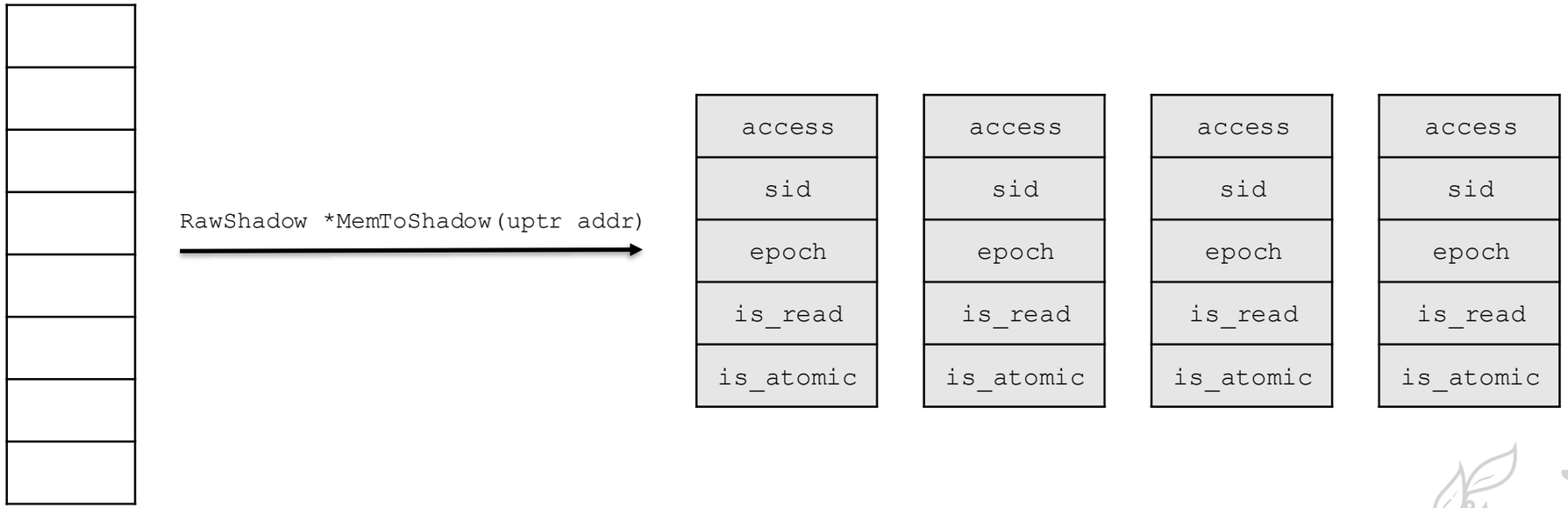


Methodology: Accuracy-Overhead Trade-offs in TSan

2. Implementation-wise

TSan keeps a record of memory accesses using *shadow memory*

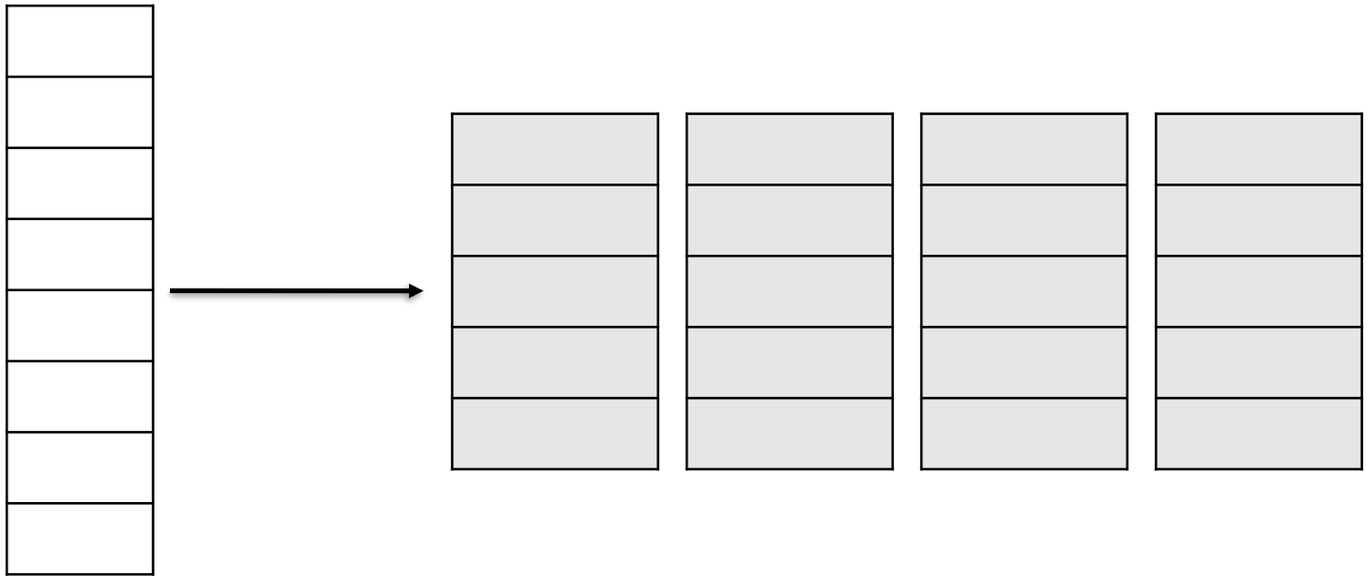
- Every 8 application bytes are mapped onto 4 *shadow values*
- Each shadow value encodes 1 memory access
- Might have to overwrite shadow values, therefore jeopardy of missing races!



Methodology: Accuracy-Overhead Trade-offs in TSan

2. Implementation-wise (Example)

Missing a race due to shadow replacement, i.e. false negative!



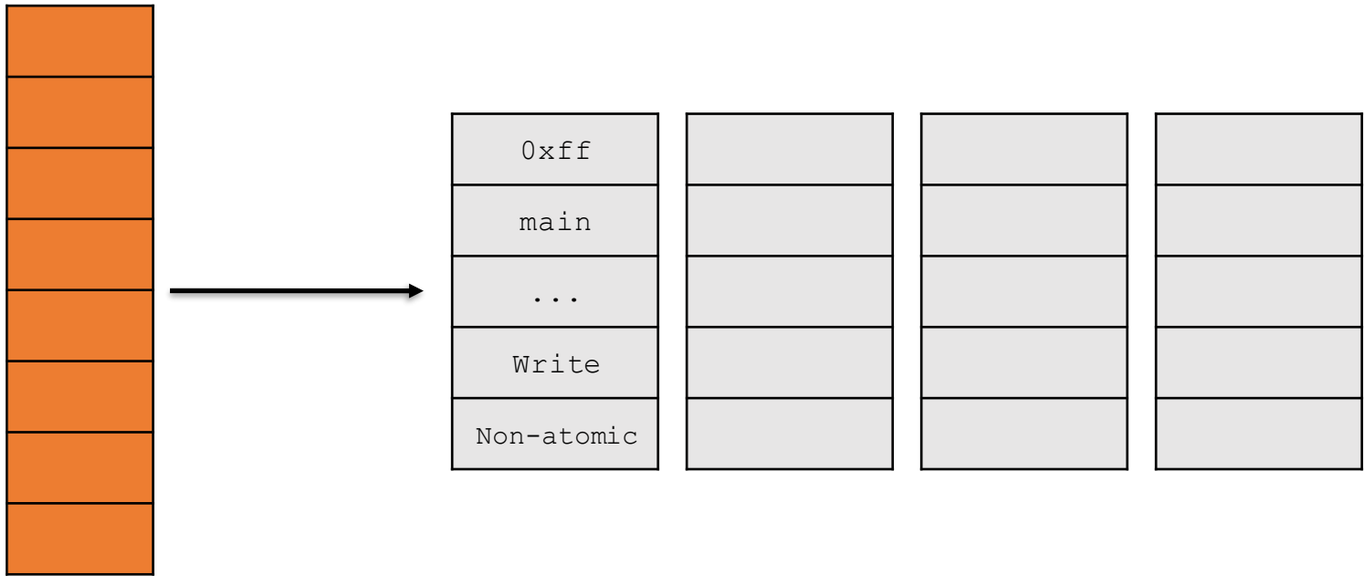
```
#include <pthread.h>
#include <stdlib.h>

pthread_mutex_t mutex;
int nb_threads = 4;

void* synch(void *ptr) {
pthread_mutex_lock(&mutex);
++ ((char *)ptr)[0];
pthread_mutex_unlock(&mutex);
return 0;
}

void* cncr_alpha(void *ptr) {
++ ((char *)ptr)[2];
++ ((char *)ptr)[3];
return 0;
}

void* cncr_beta(void *ptr) {
++ ((char *)ptr)[3];
return 0;
}

int main (void){
long *ptr = (long *)calloc(1, sizeof(*ptr));

pthread_t t[nb_threads];
pthread_mutex_init(&mutex, NULL);

pthread_create(&t[0], NULL, &synch, (void *)ptr);
pthread_create(&t[1], NULL, &cncr_alpha, (void *)ptr);
pthread_create(&t[2], NULL, &synch, (void *)ptr);
pthread_create(&t[3], NULL, &cncr_beta, (void *)ptr);


for (i = 0; i < nb_threads; ++i)
    pthread_join(t[i], NULL);

pthread_mutex_destroy(&mutex);

return 0;
}
```

Methodology: Accuracy-Overhead Trade-offs in TSan

2. Implementation-wise (Example)

Missing a race due to shadow replacement, i.e. false negative!

- Allocated 8 bytes in the main thread



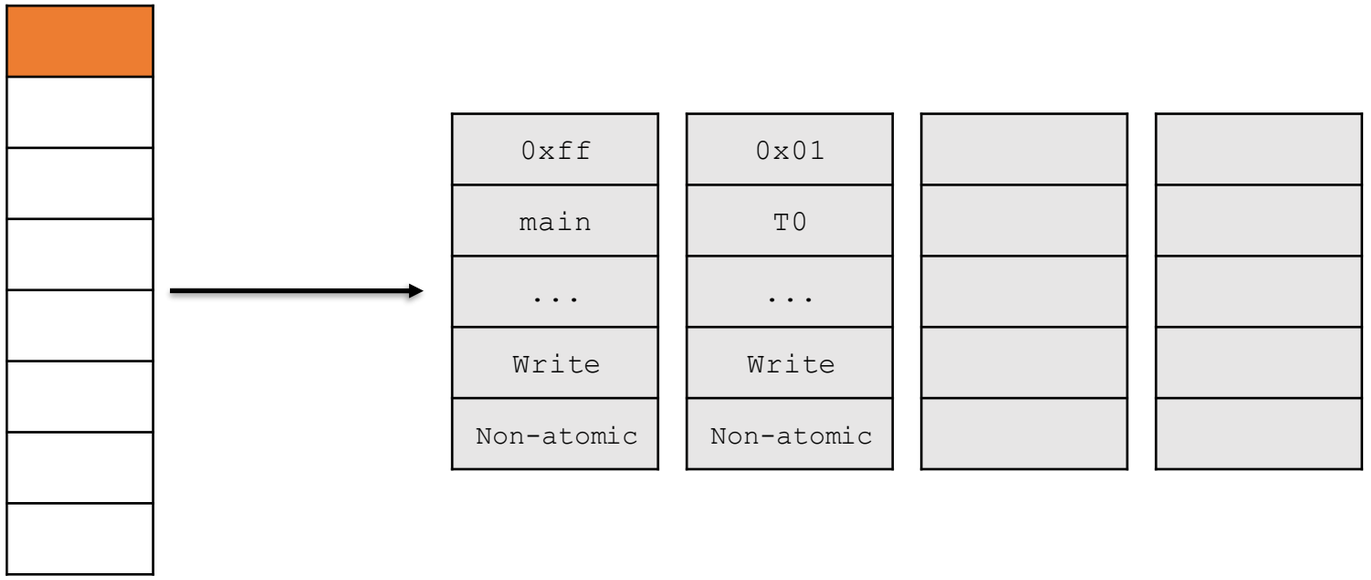

```
#include <pthread.h>
#include <stdlib.h>

pthread_mutex_t mutex;
int nb_threads = 4;

void* synch(void *ptr) {
pthread_mutex_lock(&mutex);
++ ((char *)ptr)[0];
pthread_mutex_unlock(&mutex);
return 0;
}

void* cncr_alpha(void *ptr) {
++ ((char *)ptr)[2];
++ ((char *)ptr)[3];
return 0;
}

void* cncr_beta(void *ptr) {
++ ((char *)ptr)[3];
return 0;
}

int main (void){
long *ptr = (long *)calloc(1, sizeof(*ptr));

pthread_t t[nb_threads];
pthread_mutex_init(&mutex, NULL);

pthread_create(&t[0], NULL, &synch, (void *)ptr);
pthread_create(&t[1], NULL, &cncr_alpha, (void *)ptr);
pthread_create(&t[2], NULL, &synch, (void *)ptr);
pthread_create(&t[3], NULL, &cncr_beta, (void *)ptr);


for (i = 0; i < nb_threads; ++i)
    pthread_join(t[i], NULL);

pthread_mutex_destroy(&mutex);

return 0;
}
```

Methodology: Accuracy-Overhead Trade-offs in TSan

2. Implementation-wise (Example)

Missing a race due to shadow replacement, i.e. false negative!

- Allocated 8 bytes in the main thread
- T0 accesses byte 0



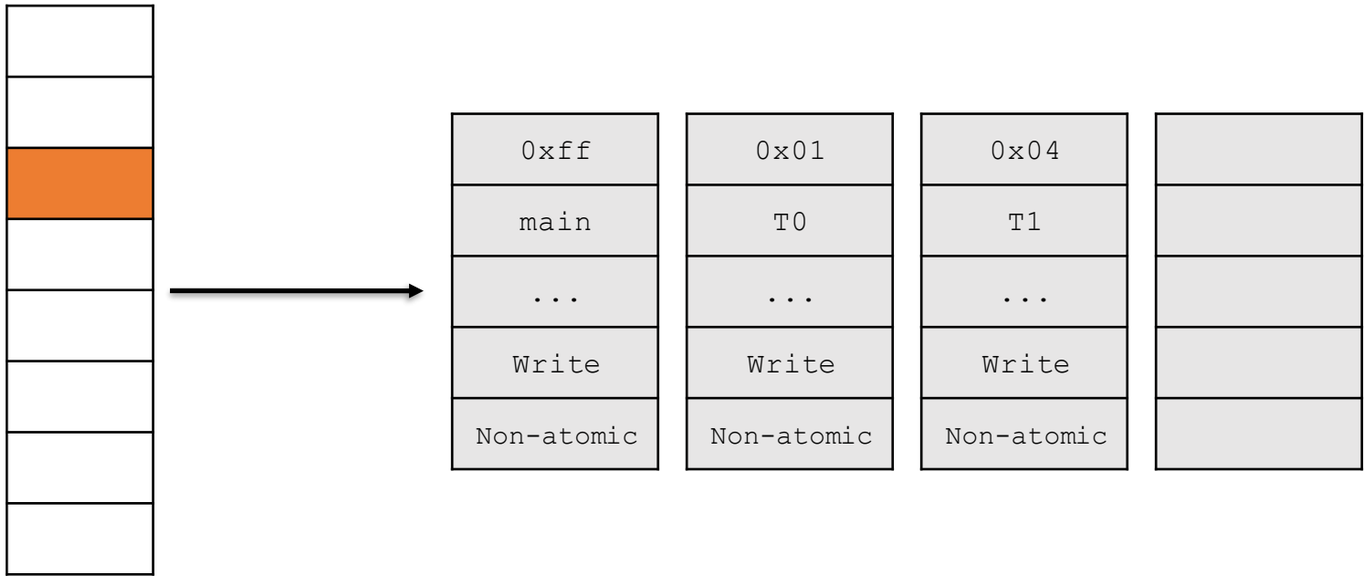

```
#include <pthread.h>
#include <stdlib.h>

pthread_mutex_t mutex;
int nb_threads = 4;

void* synch(void *ptr) {
pthread_mutex_lock(&mutex);
++ ((char *)ptr)[0];
pthread_mutex_unlock(&mutex);
return 0;
}

void* cncr_alpha(void *ptr) {
++ ((char *)ptr)[2];
++ ((char *)ptr)[3];
return 0;
}

void* cncr_beta(void *ptr) {
++ ((char *)ptr)[3];
return 0;
}

int main (void){
long *ptr = (long *)calloc(1, sizeof(*ptr));

pthread_t t[nb_threads];
pthread_mutex_init(&mutex, NULL);

pthread_create(&t[0], NULL, &synch, (void *)ptr);
pthread_create(&t[1], NULL, &cncr_alpha, (void *)ptr);
pthread_create(&t[2], NULL, &synch, (void *)ptr);
pthread_create(&t[3], NULL, &cncr_beta, (void *)ptr);


for (i = 0; i < nb_threads; ++i)
    pthread_join(t[i], NULL);

pthread_mutex_destroy(&mutex);

return 0;
}
```

Methodology: Accuracy-Overhead Trade-offs in TSan

2. Implementation-wise (Example)

Missing a race due to shadow replacement, i.e. false negative!

- Allocated 8 bytes in the main thread
- T0 accesses byte 0
- T1 accesses bytes 2



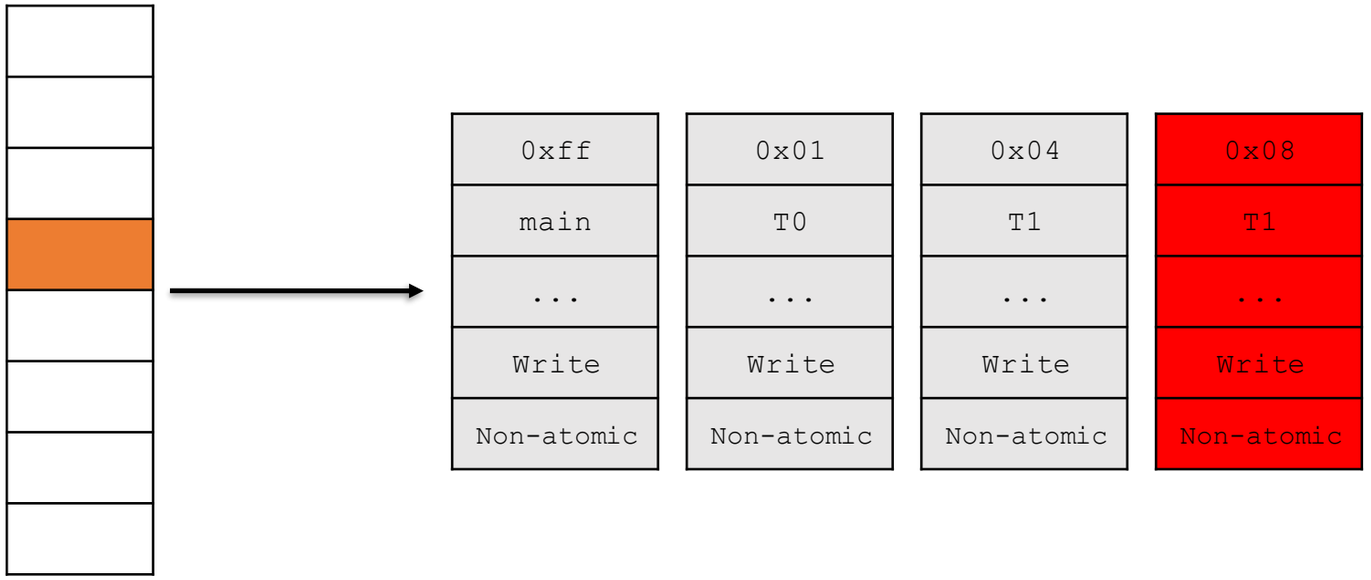

```
#include <pthread.h>
#include <stdlib.h>

pthread_mutex_t mutex;
int nb_threads = 4;

void* synch(void *ptr) {
pthread_mutex_lock(&mutex);
++ ((char *)ptr)[0];
pthread_mutex_unlock(&mutex);
return 0;
}

void* cncr_alpha(void *ptr) {
++ ((char *)ptr)[2];
++ ((char *)ptr)[3];
return 0;
}

void* cncr_beta(void *ptr) {
++ ((char *)ptr)[3];
return 0;
}

int main (void){
long *ptr = (long *)calloc(1, sizeof(*ptr));

pthread_t t[nb_threads];
pthread_mutex_init(&mutex, NULL);

pthread_create(&t[0], NULL, &synch, (void *)ptr);
pthread_create(&t[1], NULL, &cncr_alpha, (void *)ptr);
pthread_create(&t[2], NULL, &synch, (void *)ptr);
pthread_create(&t[3], NULL, &cncr_beta, (void *)ptr);


for (i = 0; i < nb_threads; ++i)
    pthread_join(t[i], NULL);

pthread_mutex_destroy(&mutex);

return 0;
}
```

Methodology: Accuracy-Overhead Trade-offs in TSan

2. Implementation-wise (Example)

Missing a race due to shadow replacement, i.e. false negative!

- Allocated 8 bytes in the main thread
- T0 accesses byte 0
- T1 accesses bytes 2 and 3




```
#include <pthread.h>
#include <stdlib.h>

pthread_mutex_t mutex;
int nb_threads = 4;

void* synch(void *ptr) {
pthread_mutex_lock(&mutex);
++ ((char *)ptr)[0];
pthread_mutex_unlock(&mutex);
return 0;
}

void* cncr_alpha(void *ptr) {
++ ((char *)ptr)[2];
++ ((char *)ptr)[3];
return 0;
}

void* cncr_beta(void *ptr) {
++ ((char *)ptr)[3];
return 0;
}

int main (void){
long *ptr = (long *)calloc(1, sizeof(*ptr));

pthread_t t[nb_threads];
pthread_mutex_init(&mutex, NULL);

pthread_create(&t[0], NULL, &synch, (void *)ptr);
pthread_create(&t[1], NULL, &cncr_alpha, (void *)ptr);
pthread_create(&t[2], NULL, &synch, (void *)ptr);
pthread_create(&t[3], NULL, &cncr_beta, (void *)ptr);


for (i = 0; i < nb_threads; ++i)
    pthread_join(t[i], NULL);

pthread_mutex_destroy(&mutex);

return 0;
}
```

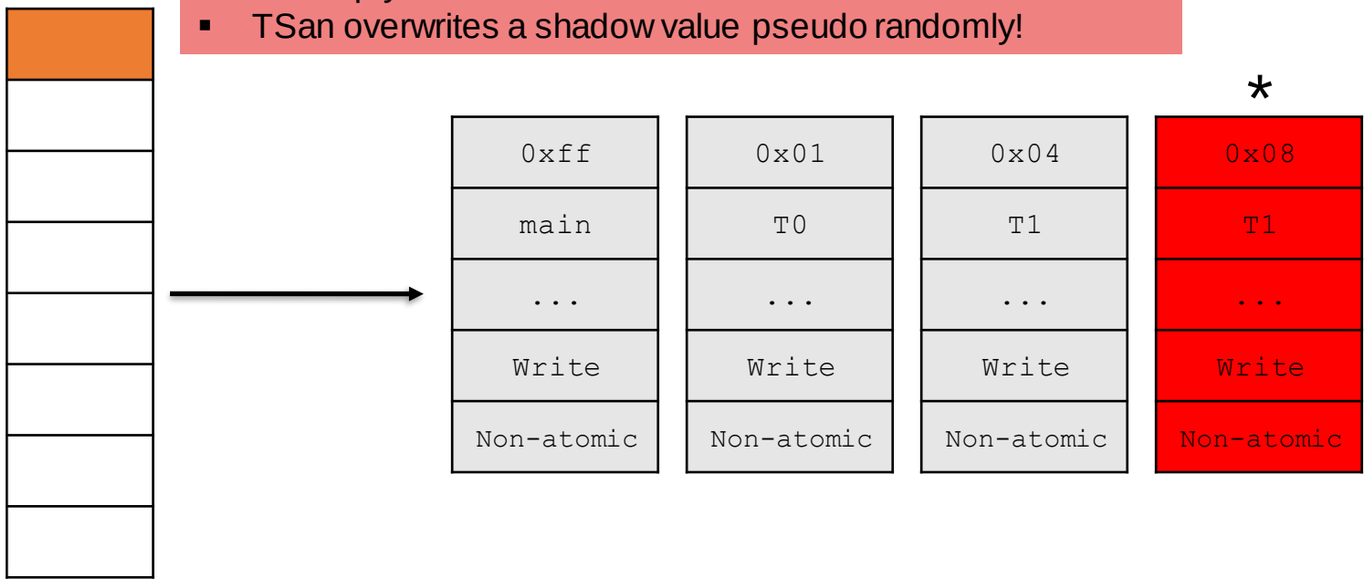
Methodology: Accuracy-Overhead Trade-offs in TSan

2. Implementation-wise (Example)

Missing a race due to shadow replacement, i.e. false negative!

- Allocated 8 bytes in the main thread
- T0 accesses byte 0
- T1 accesses bytes 2 and 3
- T2 accesses byte 0

- No empty shadow value to store current access information!
- TSan overwrites a shadow value pseudo randomly!




```
#include <pthread.h>
#include <stdlib.h>

pthread_mutex_t mutex;
int nb_threads = 4;

void* synch(void *ptr) {
pthread_mutex_lock(&mutex);
++ ((char *)ptr)[0];
pthread_mutex_unlock(&mutex);
return 0;
}

void* cncr_alpha(void *ptr) {
++ ((char *)ptr)[2];
++ ((char *)ptr)[3];
return 0;
}

void* cncr_beta(void *ptr) {
++ ((char *)ptr)[3];
return 0;
}

int main (void){
long *ptr = (long *)calloc(1, sizeof(*ptr));

pthread_t t[nb_threads];
pthread_mutex_init(&mutex, NULL);

pthread_create(&t[0], NULL, &synch, (void *)ptr);
pthread_create(&t[1], NULL, &cncr_alpha, (void *)ptr);
pthread_create(&t[2], NULL, &synch, (void *)ptr);
pthread_create(&t[3], NULL, &cncr_beta, (void *)ptr);


for (i = 0; i < nb_threads; ++i)
    pthread_join(t[i], NULL);

pthread_mutex_destroy(&mutex);

return 0;
}
```

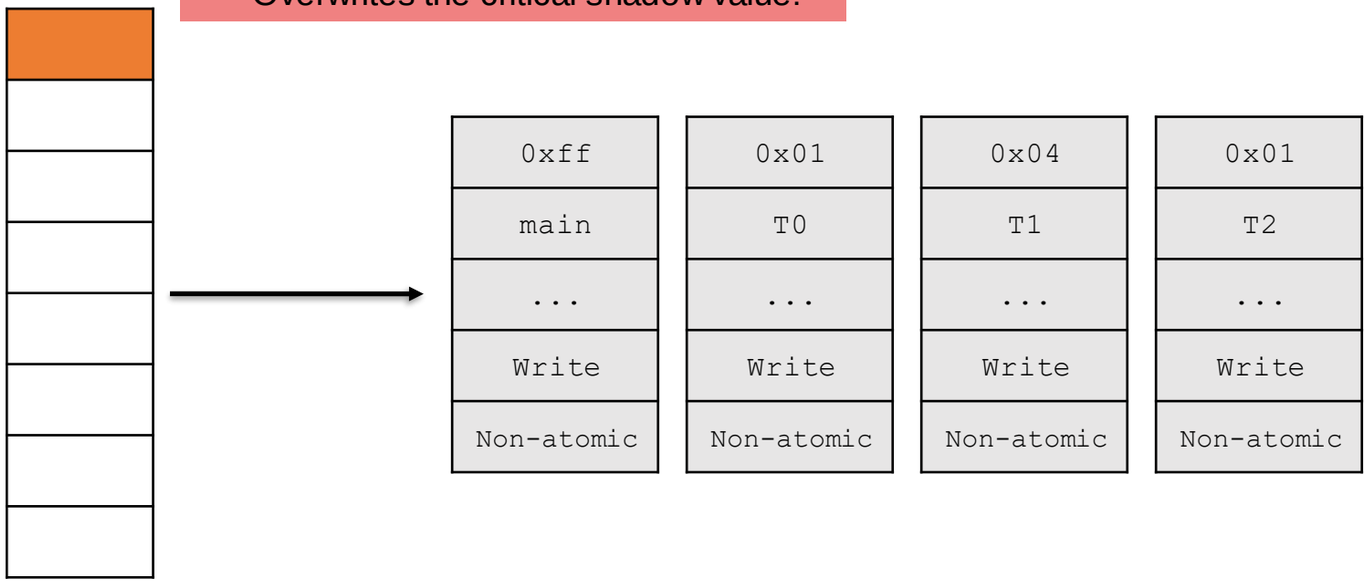
Methodology: Accuracy-Overhead Trade-offs in TSan

2. Implementation-wise (Example)

Missing a race due to shadow replacement, i.e. false negative!

- Allocated 8 bytes in the main thread
- T0 accesses byte 0
- T1 accesses bytes 2 and 3
- T2 accesses byte 0

- Overwrites the critical shadow value!



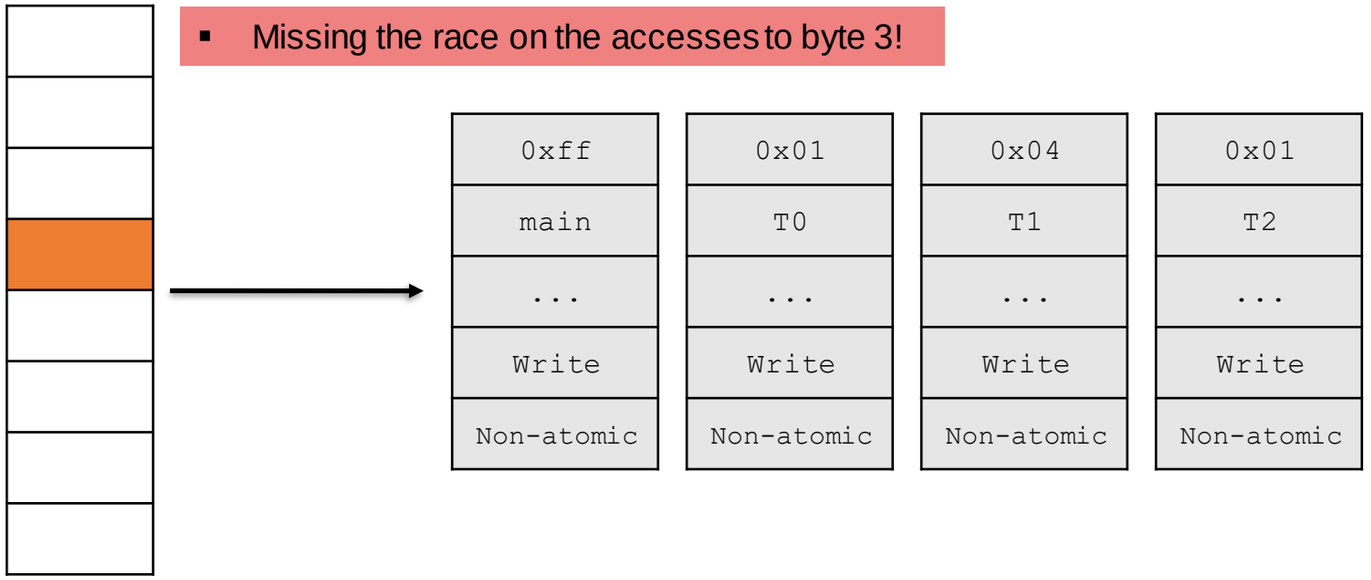

```
#include <pthread.h>
#include <stdlib.h>

pthread_mutex_t mutex;
int nb_threads = 4;

void* synch(void *ptr) {
pthread_mutex_lock(&mutex);
++ ((char *)ptr)[0];
pthread_mutex_unlock(&mutex);
return 0;
}

void* cncr_alpha(void *ptr) {
++ ((char *)ptr)[2];
++ ((char *)ptr)[3];
return 0;
}

void* cncr_beta(void *ptr) {
++ ((char *)ptr)[3];
return 0;
}

int main (void){
long *ptr = (long *)calloc(1, sizeof(*ptr));

pthread_t t[nb_threads];
pthread_mutex_init(&mutex, NULL);

pthread_create(&t[0], NULL, &synch, (void *)ptr);
pthread_create(&t[1], NULL, &cncr_alpha, (void *)ptr);
pthread_create(&t[2], NULL, &synch, (void *)ptr);
pthread_create(&t[3], NULL, &cncr_beta, (void *)ptr);


for (i = 0; i < nb_threads; ++i)
    pthread_join(t[i], NULL);

pthread_mutex_destroy(&mutex);

return 0;
}
```

Methodology: Accuracy-Overhead Trade-offs in TSan

2. Implementation-wise (Example)

Missing a race due to shadow replacement, i.e. false negative!

- Allocated 8 bytes in the main thread
- T0 accesses byte 0
- T1 accesses bytes 2 and 3
- T2 accesses byte 0
- T3 accesses byte 3




```
#include <pthread.h>
#include <stdlib.h>

pthread_mutex_t mutex;
int nb_threads = 4;

void* synch(void *ptr) {
pthread_mutex_lock(&mutex);
++ ((char *)ptr)[0];
pthread_mutex_unlock(&mutex);
return 0;
}

void* cncr_alpha(void *ptr) {
++ ((char *)ptr)[2];
++ ((char *)ptr)[3];
return 0;
}

void* cncr_beta(void *ptr) {
++ ((char *)ptr)[3];
return 0;
}

int main (void){
long *ptr = (long *)calloc(1, sizeof(*ptr));

pthread_t t[nb_threads];
pthread_mutex_init(&mutex, NULL);

pthread_create(&t[0], NULL, &synch, (void *)ptr);
pthread_create(&t[1], NULL, &cncr_alpha, (void *)ptr);
pthread_create(&t[2], NULL, &synch, (void *)ptr);
pthread_create(&t[3], NULL, &cncr_beta, (void *)ptr);


for (i = 0; i < nb_threads; ++i)
    pthread_join(t[i], NULL);

pthread_mutex_destroy(&mutex);

return 0;
}
```

Methodology: Preliminary Results

Benchmark	#Threads	R/RF	Object Type	Runtime Overhead		Memory Overhead		Post-mortem Overhead
				TSan	Proposed *	TSan	Proposed **	Proposed ***
#1	2	Race-free	Heap	6.5×	1.1×	7.1×	0%	34%
#2	3	Data Race	Global	8.8×	1.3×	7.7×	0%	28%
#3	3	Data Race	Heap	9.4×	1.3×	7.8×	0%	29%

- Three C++ micro benchmarks
- No false positive or false negative report
- The proposed post-mortem analyzer is not complete and optimized yet

* 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

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
  - Promising preliminary results in comparison with TSan
- Future Work:
  - Exploit the potential of static code analysis to specify memory accesses that do not affect the correctness of data race detection if not instrumented
  - Applying a similar idea to other google sanitizers

Questions?

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