CS 636: Testing of Concurrent Programs

Swarnendu Biswas

Department of Computer Science and Engineering,
Indian Institute of Technology Kanpur

Sem 2025-26-II

Evaluating Concurrent Programs

Functional correctness

- Does the application compute what it is supposed to do?
- Check for concurrency errors such as atomicity violations, order violations, sequential consistency violations, deadlocks, and livelocks

Performance correctness

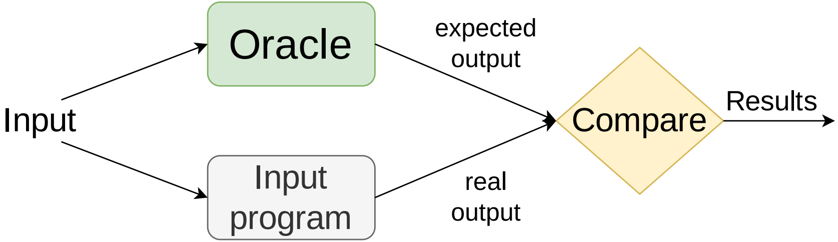
- Does the application meet the performance requirements?
- Difficult to detect performance bottlenecks because of no failure symptoms
- Check for any performance regressions

Ideas to Ensure Correctness of Concurrent Programs

- Programming language features ensure bad things cannot happen by design (e.g., DPJ†)
  - Restricts the power and expressiveness of the language
- Design algorithms that are resilient to errors
  - Limits the kind of data structures that you can use
- Testing cannot guarantee correctness, usually a “best effort” strategy
  + Places no restrictions on the application

† Deterministic Parallel Java

Software Testing



> 50% of my company employees are testers, and the rest spend 50% of their time testing!
>
> – Bill Gates, 1995.

Testing Concurrent Programs is Hard!

- Nondeterminism is everywhere
  - May be inherent in the application or can be due to inputs or interleavings
  - Large space of all possible thread interleavings
- Only specific thread interleavings may expose a concurrency bug (often called “Heisenbugs”)
  - Random or naïve testing can often miss such errors
- Even when found, errors are hard to debug
  - Usually no repeatable trace, just retrying the execution may not reproduce the error if it is rare
  - Debugging with `print` statements may actually change the desired buggy interleaving
  - Source of the bug may be far away from where it manifests
- Huge productivity problem
  - Developers and testers often spend weeks chasing after a single Heisenbug!

High-level Requirements for Testing Concurrent Programs

- Test code, test inputs, and test oracles – a test harness
- A deterministic schedule may be needed to validate with the oracles
- Associated notion of coverage – test as many interleavings as possible

Possibilities in Testing Concurrent Programs

1. Exhaustively explore all possible interleavings
2. Deterministic testing
   - Controls thread scheduling decisions during execution and systematically explores interleavings
   - Depends on a deterministic scheduler
   - Nondeterminism could still be there due to inputs
3. Nondeterministic "best effort" testing
   - Run the program for some time and hope for the best
   - Naïve and inefficient
4. Stress testing
   - Launch more threads than processors so that only a few threads are running at a time
   - Try to decrease predictability in thread interleavings
5. Noise injection
   - Introduce random perturbations during execution
   - Should not introduce false positives

Alternatives to Testing

- Reason about correctness without running the program
  - Static analysis, Theorem proving, and Model checking
- Model checking checks whether a system model satisfies the given specification
  - Suffers from state explosion problem
  - Uses partial order reduction to deal with the state space problem
  - Use is limited to only critical portions of the program
- — Sophisticated static analysis and model checking do not scale well

- Trying to prove programs correct requires a formal or mathematical characterization of the programs behavior
  - — Very difficult for large systems since there are a lot of unknowns
    - For example, how do you model VM behavior like JIT compilation and GC?
  - Use is often limited to safety-critical software like integrated circuit design

Address Nondeterminism

- Enforce the correct schedule that needs to be executed
  - Deterministic execution: record and replay
- Explore all possible schedules
  - Stateful exploration
    - Model the program state at each step and use backtrack and state comparison to explore new schedules
    - Advantage is it can merge same states, alleviating the state space explosion problem
    - Java PathFinder is the state-of-art tool
  - Stateless exploration
    - Does not maintain program state
    - Each schedule maintains all the choices made during execution
    - Need to start from the beginning to execute other schedules
    - Each run is faster than stateful exploration, but possibly has more schedules to explore

Software Testing vs Concurrency Testing

Software Testing

- Broad area of work which considers the overall quality of the software along with the integrated engineering processes
  - Lots of paradigms, processes, and testing levels

Concurrency Testing

- The context that we will be discussing has more narrow focus
  - Try to improve bug detection coverage of concurrent programs
  - Mostly carried out by the developers themselves during unit testing

Software Testing vs Concurrency Testing

Software Testing

- Broad area of work which considers the overall quality of the software along with the integrated engineering processes

Concurrency Testing

- The context that we will be discussing has more narrow focus
  - Try to improve bug detection coverage of concurrent programs
  - ...velopers ...ng

- A concurrency bug manifests on a strict subset of possible schedules
  - Bugs that manifest in all schedules are not concurrency bugs
- The problem of concurrency testing is to find those schedules that can trigger these bugs

Current Practice in Concurrency Testing

- Concurrency testing is often delegated to random testing and stress testing
- Example: Test a concurrent queue implementation
  - Create numerous threads performing queue operations
  - Run for several hours
  - Randomly perturb the execution
- Stressing the system increases the likelihood of rare interleavings
  - Makes any error found hard to debug

Performance Testing

- No good tools for predicting system performance
  - Check for latency, resource consumption

- Other considerations
  - Garbage Collection (GC) may take arbitrarily long and may be triggered at random points
    - Either turn off GC or design tests that invoke multiple GCs so that it can be averaged out
  - Dynamic compilation with JIT compiler
    - Methods compiled and time taken impacts the measured time of the program
    - Mixing interpretation and JIT is random
    - Fix which methods are going to be compiled beforehand and only compile those at runtime

Related Directions

- Techniques to expose concurrency bugs[§†]
- Techniques to generate test cases (inputs) to trigger concurrency bugs
- Technique to automatically fix concurrency bugs [‡] [¶]
- ...

[§] D. Wolff et al. Greybox Fuzzing for Concurrency Testing. ASPLOS'24.

[†] H. Zhao et a. Selectively Uniform Concurrency Testing. ASPLOS'25.

[‡] G. Jin et al. Automated Atomicity-Violation Fixing. PLDI'11.

[¶] H. Lin et al. PFix: Fixing Concurrency Bugs Based on Memory Access Patterns. ASE'18.

Finding Concurrency Bugs Based on Code Patterns

Insights Related to Concurrency Bugs

- — Programmers make simple mistakes because of a tendency to think sequentially
- — Natural tendency is to under-synchronize in pursuit of performance
  - Misconception that shared-memory synchronization is slow[§]
  - Lots of research to optimize the common case of low contention
- Indirect influence of the programming toolchain
  - \+ Writing threaded code with Java is comparatively easier
  - — Java gives **limited** guarantees with improperly synchronized code unlike C and C++
    - You get type and memory safety, so why bother!!!

[§] J. Preshing. Locks Aren't Slow; Lock Contention Is.

Overview of SpotBugs[†]

- Open-source static analysis tool for Java
- Goal is to use simple program analysis to find common patterns that indicate errors
  - Similar in spirit to automated code reviews
  - As such there can be both false negatives and false positives
  - Tries to minimize false positives using heuristics but cannot eliminate them completely
- Potential errors are classified into levels depending on estimated impact
- There is also a notion of confidence along with each reported error
- Lot of plugins are available for tools like Eclipse, IntelliJ, Ant, and Maven
- SpotBugs is a successor of FindBugs[¶]

[¶] D. Hovemeyer and W. Pugh. Finding Concurrency Bugs in Java. PODC Workshop on Concurrency and Synchronization in Java Programs, 2004.

[†] SpotBugs: Find bugs in Java Programs

Examples of Patterns Used in SpotBugs

- Synchronized set method, unsynchronized get method
- Finalizer method only nulling out fields
- Object pair operations with lock on only one object (e.g., `equals()` method)
- Double-checked locking

```java
static SomeKls field;
static SomeKls createSingleton() {
  if (field == null)
    synchronized (lock) {
      if (field == null) {
        SomeKls obj = new SomeKls();
        field = obj;
      }
    }
    return field;
}
```

Bug descriptions

Examples of Patterns Used in SpotBugs

- Unconditional wait
- Wait and notify without holding lock on the object, or two locks held while waiting
  - Intraprocedural analysis to identify lock scopes
- Spin wait on non-volatile data
- If overriding `equals()`, then `hashcode()` should be overridden too

```
if (!book.isReady()) {
  synchronized (book) {
    book.wait();
  }
}
```

```
// non-volatile field
while (listLock) {}
```

Bug descriptions

Patterns Used in SpotBugs

Over 400 bug patterns divided into different categories

- All accesses to fields of a thread-safe class should be guarded with locks, otherwise are reported as bugs
  - Reduce false positives —- ignore accesses in constructors and finalizers, ignore volatiles, final, and non-final public fields
- Ranks reports based on access frequency
  - 25% or fewer unsynchronized accesses is classified as medium to high priority
  - 25-50% unsynchronized accesses are classified as low priority

Bug descriptions

Relevance of FindBugs/SpotBugs

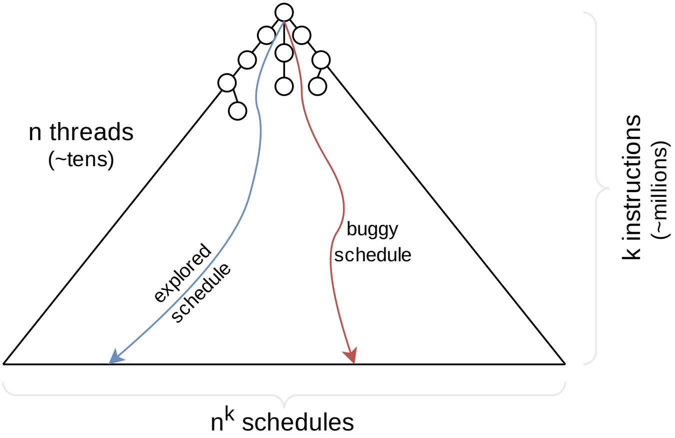
- An early work (∼2004) that was very effective in pointing out errors in real applications like the Java libraries
  - Implementation is still being actively maintained

```java
// From Eclipse 3.5RC3:
// org.eclipse.update.internal.ui.views.FeatureStateAction:

if (adapters == null && adapters.length == 0)
  return;

// First seen in Eclipse 3.2
// In practice, adapters is probably never null
```

Probabilistic Concurrency Testing

Exposing a Concurrency Bug with Random Testing

- Exposing a concurrency bug requires reproducing the correct interleaving
- No algorithm can find the bug with a probably greater than $\frac{1}{n^k}$



Debugging with Randomized Scheduling

Consider a naïve randomized scheduler that flips a coin in each step to decide which thread to schedule next

Thread 1

```
assert(b != 0);
step(1);
step(2);
...
...
step(m);
a = 0;
```

Thread 2

```
assert(a != 0);
step(1);
step(2);
...
...
step(n);
b = 0;
```

Categorizing Concurrency Bugs

Bug depth is the number of ordering constraints that need to be satisfied to trigger the bug

Thread 1

```
void init(...) {
  ...
  ...
  ...
  mThread = PR_CreateThread(mMain, ...);
  ...
}
```

Thread 2

```
...
void mMain() {
  mState=mThread->State;
  ...
}
```

Mozilla: nsthread.cpp

A Bug of Depth 1

Parent

```
A: ...
B: fork(child);
C: p = malloc();
D: ...
E: ...
```

Child

```
F: ...
G: do_init();
H: p->f++;
I: ...
J: ...
```

Possible Schedules

- ABCDEFGHIJ ✓
- ABFGHCDEIJ ✗
- ABFGCDEHIJ ✓
- ABFGCHDEIJ ✓
- ABFGHIJCDE ✗
- ...

A Bug of Depth 2

Parent

```
A: ...
B: p = malloc();
C: fork(child);
D: ...
E: if (p != NULL)
F:   p->f++;
G:
```

Child

```
H: ...
I: p = NULL;
J: ...
```

Possible Schedules

- ABCDEFGHIJ ✓
- ABCDEHIJFG ✗
- ABCHIDEGJ ✓
- ABCDHEFIJG ✓
- ABCHDEIJFG ✗
- ...

Another Bug of Depth 2

Parent

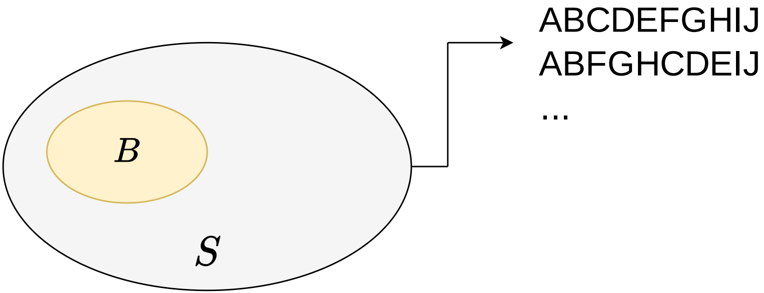
```
A: ...
B: lock(m);
C: ...
D: lock(n);
E: ...
```

Child

```
F: ...
G: lock(n);
H: ...
I: lock(m);
J: ...
```

What is Bug Depth?

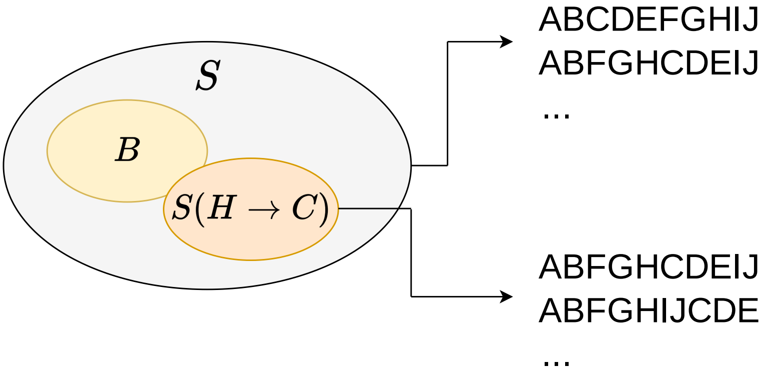
- A system is defined by its set of executions S
- Each execution is a sequence of labeled events
- A concurrency bug B is some **strict** subset of S



M. Musuvathi. Randomized Algorithms for Concurrency Testing. CONCUR, 2017.

What is Bug Depth?

- An ordering constraint c is a pair of events $c = (a \rightarrow b)$
- A schedule $s \in S$ satisfies $(a \rightarrow b)$ if a occurs before b in s
- Let $S(c_1, c_2, \ldots, c_d)$ be the set of schedules that satisfy constraints $c_1, c_2, \ldots, c_d$

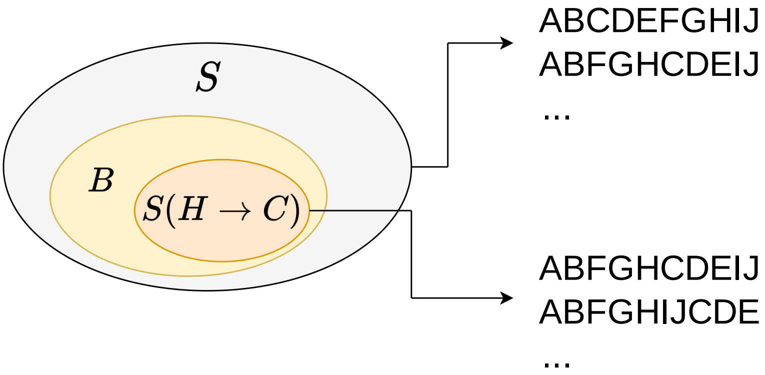


What is Bug Depth?

A bug depth is d if there exists constraints $c_1, c_2, \ldots, c_d$ such that

$$S(c_1, c_2, \ldots, c_d) \subseteq B$$

and d is the smallest such number for B

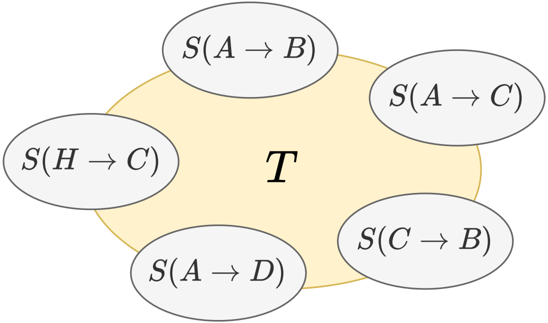


Finding All Bugs of Depth d

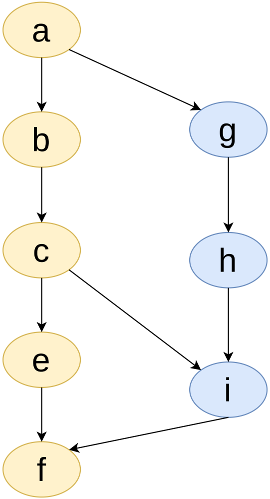
- A set of schedules T covers all bugs of depth d if

$$\forall c_1, c_2, \ldots, c_d : S(c_1, c_2, \ldots, c_d) \cap T \neq \phi$$

- The coverage problem is to find the smallest such T

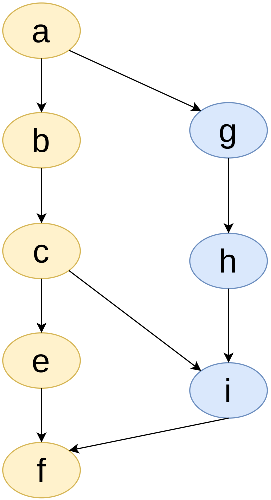


Concurrent Interleavings when $d = 1$

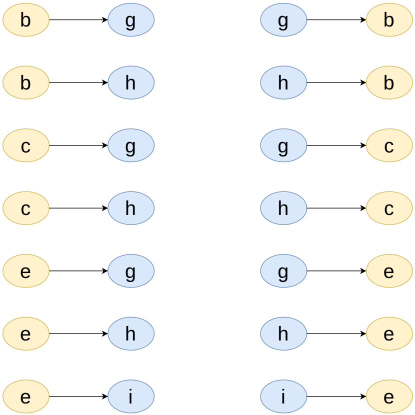


Which pair of operations are concurrent?

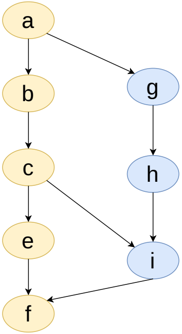
Concurrent Interleavings when $d = 1$



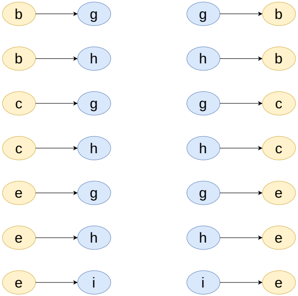
Need to cover all



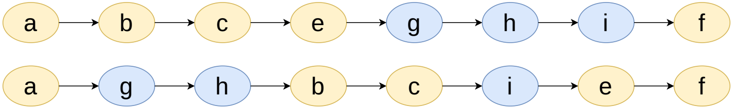
Concurrent Interleavings when $d = 1$



Need to cover all



Two interleavings are sufficient!



Concurrency Bugs and Bug Depth

- Most concurrency bugs are usually of **low** depth

Order violations	depth 1 (or 2 in presence of control flow)
Atomicity violations	depth 2
Deadlocks	depth 2 if 2 threads are involved, depth *n* if *n* threads are involved

- Bugs with greater depth are harder to expose

A Bug of Depth 2

Main Thread

```
...
free(mutex);

exit(0);
...
```

Filewriter Thread

```
...

mutex.unlock();

...
```

S. Burckhardt et al. A Randomized Scheduler with Probabilistic Guarantees of Finding Bugs. ASPLOS, 2010.

An Ordering Bug of Depth 2

Main Thread

```
...
init = true;
t = new T();
...
...
```

Filewriter Thread

```
...
...
if (init)
  t->state = 1;
...
```

Presence of control dependence may complicate the interleaving

S. Burckhardt et al. A Randomized Scheduler with Probabilistic Guarantees of Finding Bugs. ASPLOS, 2010.

PCT: Probabilistic Concurrency Testing

- PCT is an intelligent randomized scheduler for finding concurrency bugs
- PCT aims to correctly schedule instructions relevant to expose a bug, irrelevant instructions are ignored to reduce the search space
- Provides probabilistic guarantees to expose bugs
  - Every run finds every bug with nontrivial probability
  - Repeated test runs increases the chance of finding a bug

S. Burckhardt et al. A Randomized Scheduler with Probabilistic Guarantees of Finding Bugs. ASPLOS, 2010.

PCT's Randomized Scheduler

- User-level scheduler is randomized and priority-based
  - Every thread has a priority, lower number indicates lower prioritie
- Only one thread is scheduled to execute at each step
- Low priority threads are scheduled only when higher-priority threads are blocked

- A dynamic execution has a few priority change points
  - Priority change points have fixed priorities assigned
  - A thread that reaches a change point will inherit the priority of the change point

PCT Algorithm

Input n threads, k instructions, and d priority change points

Steps

(i) Assign n priority values $d, d+1, \ldots, d+n-1$ randomly to the n threads

(ii) Pick $d-1$ random priority change points from the k instructions. Each change point $k_i, 1 \leq i < d$, has an associated priority of i.

(iii) Schedule threads based on their priorities. The highest priority thread that is enabled runs for one step.

(iv) When a thread reaches change point k_i, change the priority of that thread to i

Assumptions in PCT

Higher priority threads run faster

An ordering constraint ($a \rightarrow b$) will be met if a is executed by a higher priority thread

How PCT Works?

initial thread priority

Thread 1

1

```
...
t = new T();
...
```

Thread 2

2

```
...

if (t->state == 1)
  ...
```

How PCT Works?

Thread 1 2

```
...
x = NULL;
...
```

Thread 2 3

priority change point

1

```
...
if (x != NULL)

  x->print();
```

How PCT Works?

Thread 1 3

```
...
lock(a);
...
lock(b);
...
```

1

Thread 2 2

```
...
lock(b);
...
lock(a);
...
```

Issues to Consider in PCT

- Does not reuse OS thread priorities
  - PCT implements a user-level scheduler instead
  - Needs to force higher priority threads to run faster
- Consider priority inversion in presence of multiple threads
  - Higher priority thread may be blocked for a resource owned by a lower priority thread violating PCT's assumptions
    - Assume that Thread 2 needs to run before Thread 1 to expose a bug
    - Thread 1 has a lower priority than Thread 2, but Thread 2 is blocked on a resource held by Thread 3 which has the lowest priority
  - But there will be other schedules where the priorities will be in the correct order with probability $\frac{1}{n}$
- Ensure starvation freedom
  - Repeatedly slowing down the low-priority thread can cause starvation or timeout
  - Higher priority threads may wait in a spin loop for a lower priority thread
  - Uses heuristics to identify and resolve such situations

Effectiveness of PCT

- Probability of finding any bug with depth d in PCT is not less than $\frac{1}{nk^{(d-1)}}$
  - Contrast with the probability of naïve random testing which is $\frac{1}{n^k}$
- If $d = 1$ or $d = 2$ (common cases), then probabilities of finding a bug is $\frac{1}{n}$ and $\frac{1}{nk}$, respectively
- PCT is empirically expected to do better than the worst-case bound

Why?

Effectiveness of PCT

- Probability of finding any bug with depth d in PCT is $\frac{1}{nk^{(d-1)}}$
  - Contrast with the probability of naïve random testing which is $\frac{1}{n^k}$
- If $d = 1$ or $d = 2$ (common cases), then probabilities of finding a bug is $\frac{1}{n}$ and $\frac{1}{nk}$, resp[ectively]
- PCT

- Good enough to have the priority change point on one from a set of instructions, need not be exact
- Multiple ways to trigger a bug (e.g., symmetric case in deadlocks)
- Buggy code can be repeated multiple times in a program/test

Extensions of PCT

- PCT runs only a single thread at a time
  - Does not utilize multicore hardware, incurs large slowdowns
- PPCT: Parallel PCT
  - Insight: Need to control the schedule of only d threads to expose a bug of depth d
  - Partitions threads into high ($> d$) and low priority
  - Runs threads with higher priority parallelly, size of the lower priority set is bounded by d
  - PCT serializes all threads, PPCT serializes only the low priority threads

S. Nagarakatte et al. Multicore Acceleration of Priority-Based Schedulers for Concurrency Bug Detection. PLDI, 2012.

PPCT Algorithm

Input n threads, k instructions, and d priority change points

Steps

1. Pick a random thread and assign it a priority d. Insert the thread in a low priority set L. Insert all other threads into a high priority set H.
2. Pick $d-1$ random priority change points from the k instructions. Each change point $k_i, 1 \leq i < d$ has an associated priority of i.
3. At each scheduling step, schedule any non-blocked thread in H. If H is empty or if all threads in H are blocked, then schedule the highest priority thread in L.
4. When a thread reaches change point k_i, change the priority of that thread to i and insert in L.

CHESS: Systematic Schedule Exploration

What have we learnt so far?

- Systematic schedule exploration enumerates all possible thread interleavings
  - Does not scale
- PCT and PPCT argued in favor of intelligent randomized testing

CHESS performs systematic schedule exploration

M. Musuvathi et al. Finding and Reproducing Heisenbugs in Concurrent Programs. OSDI 2008.

Traditional Testing

```
testStartup();
while (true) {
  runTestScenario();
  if (*some condition*)
    break;
}
testShutdown();
```

What is required for systematic exploration?

- Suppose you have two threads contending on a lock
- Systematic exploration should explore both schedules — one where each thread wins the lock first

Basically capture all nondeterministic choices

Why Track Nondeterminism?

Capture all sources of nondeterminism

- For example, input, environment, interleaving, and other sources like compiler and hardware reordering

Allows exploring these nondeterministic choices

Required for reliably reproducing errors

Input Nondeterminism

- Environment data can affect program execution
  - User can provide different inputs or the program can receive network packets with different contents
  - Nondeterministic functions like `gettimeofday()` and `random()`

- Idea: Use “record and replay” techniques
  - Two phases — a record phase and a replay phase
  - Which phase is usually more expensive, record or replay?

Capturing Input Nondeterminism in CHESS

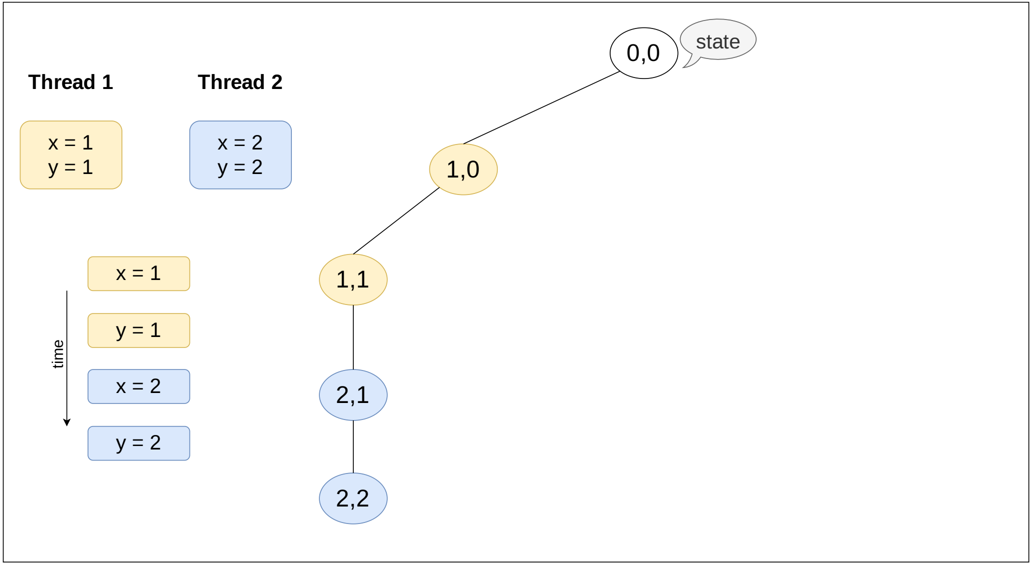
- CHESS is not a typical record-and-replay system
- Relies on the test setup to provide deterministic inputs
- Records a few nondeterministic events like current time, processor and thread ID mapping, and random numbers

M. Musuvathi et al. Finding and Reproducing Heisenbugs in Concurrent Programs. OSDI 2008.

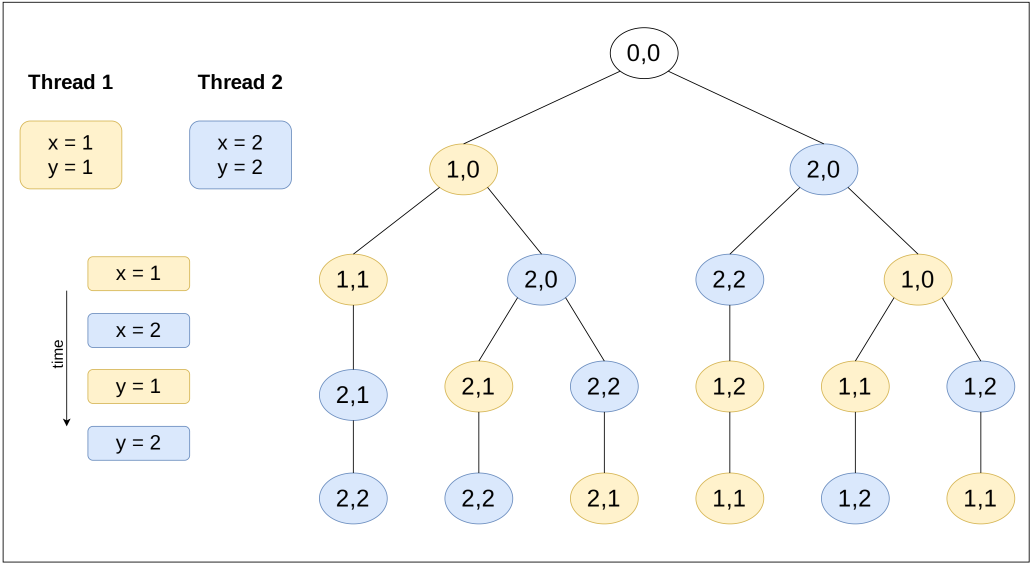
Concurrent Executions are Nondeterministic

Thread 1	Thread 2
x = 1 y = 1	x = 2 y = 2

Concurrent Executions are Nondeterministic



Concurrent Executions are Nondeterministic



Scheduling Nondeterminism

Interleaving nondeterminism

- Threads can race to access shared variables or monitors
- OS can preempt threads at arbitrary points

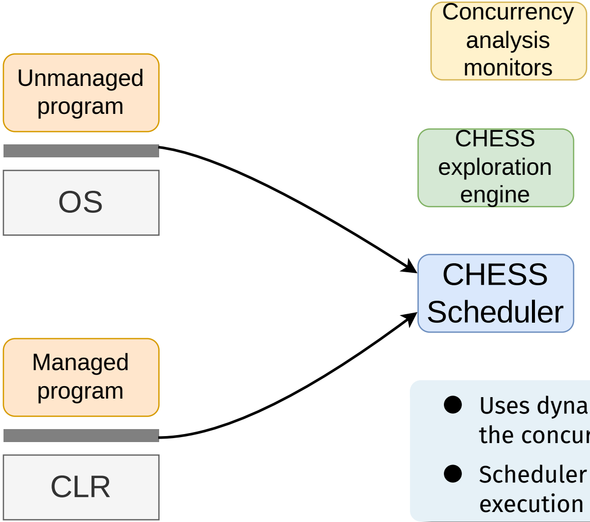
Timing nondeterminism

- Timers can fire in different orders
- Sleeping threads wake up at arbitrary times in the future
- Asynchronous calls complete at arbitrary times in the future

CHESS in a nutshell

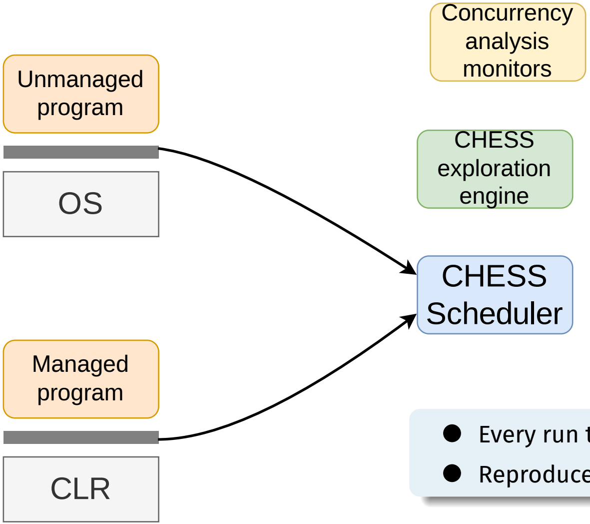
- User-mode scheduler — controls all scheduler nondeterminism
- Provides systematic overage of all thread interleavings
  - Every program run takes a different thread interleaving
- CHESS is precise, does not introduce new behaviors
- Provides replay capability for easy debugging
  - Reproduce the interleaving for every run

CHESS Architecture



- Uses dynamic binary instrumentation to intercept calls to the concurrency library
- Scheduler captures the happens-before graph of the execution

CHESS Architecture



- Every run takes a different interleaving
- Reproduce the interleaving for every run

Interleaving Nondeterminism

```
balance = 100;
```

Deposit Thread

```
void Deposit100() {
  EnterCriticalSection(&cs);
  balance += 100;
  LeaveCriticalSection(&cs);
}
```

Withdrawal Thread

```
void Withdraw100() {
  EnterCriticalSection(&cs);
  int t = balance;
  LeaveCriticalSection(&cs);

  EnterCriticalSection(&cs);
  balance = t - 100;
  LeaveCriticalSection(&cs);
}
```

```
assert(balance == 100);
```

Invoke the Scheduler at Preemption Points

```
balance = 100;
```

Deposit Thread

```
void Deposit100() {
  ChessSchedule();
  EnterCriticalSection(&cs);
  balance += 100;
  ChessSchedule();
  LeaveCriticalSection(&cs);
}
```

Withdrawal Thread

```
void Withdraw100() {
  ChessSchedule();
  EnterCriticalSection(&cs);
  int t = balance;
  ChessSchedule();
  LeaveCriticalSection(&cs);
  ChessSchedule();
  EnterCriticalSection(&cs);
  balance = t - 100;
  ChessSchedule();
  LeaveCriticalSection(&cs);
}
```

Each call is a potential preemption point

```
assert(balance == 100);
```

Insert Predictable Delays with Additional Synchronization

Deposit Thread

```
void Deposit100() {
  waitEvent(e1);
  EnterCriticalSection(&cs);
  balance += 100;
  LeaveCriticalSection(&cs);
  setEvent(e2);
}
```

Withdrawal Thread

```
void Withdraw100() {
  EnterCriticalSection(&cs);
  int t = balance;
  LeaveCriticalSection(&cs);
  setEvent(e1);
  waitEvent(e2);
  EnterCriticalSection(&cs);
  balance = t - 100;
  LeaveCriticalSection(&cs);
}
```

Blindly Inserting Delays can lead to Deadlocks!

Deposit Thread

```
void Deposit100() {
  EnterCriticalSection(&cs);
  balance += 100;
  waitEvent(e1);
  LeaveCriticalSection(&cs);
}
```

Withdrawal Thread

```
void Withdraw100() {
  EnterCriticalSection(&cs);
  int t = balance;
  LeaveCriticalSection(&cs);
  setEvent(e1);
  EnterCriticalSection(&cs);
  balance = t - 100;
  LeaveCriticalSection(&cs);
}
```

CHESS Scheduler Basics

- CHESS is a non-preemptive, fair, round-robin and priority-based, starvation-free scheduler
  - Executes chunks of code atomically
- Scheduler basically captures the happens-before graph for the execution
- Each graph node tracks threads, synchronization resources, and the operations, and whether tasks are enabled or disabled
- Introduces an event per thread, every thread blocks on its event
- The scheduler wakes one thread at a time by enabling the corresponding event
- The scheduler does not wake up a disabled thread
  - Need to know when a thread can make progress
  - Synchronization wrappers provide this information
- The scheduler has to pick one of the enabled threads
  - The exploration engine decides for the scheduler

CHESS Scheduler Basics

Three Steps

- **Record**: Schedules a thread till the thread yields
- **Replay**: Replays a sequence of scheduling choices from a trace file
- **Search**: Uses the enabled information at each schedule point to determine the scheduler for the next iteration

Traditional Testing vs CHESS

Traditional Testing

```
testStartup();
while (true) {

  runTestScenario();


  if (some condition)
    break;

}
testShutdown();
```

CHESS

```
testStartup();
while (true) {

  runTestScenario();


  if (some condition)
    break;

}
testShutdown();
```

replay

record

search

Preemption bounding

- Systematically inserts a small number of preemptions
- Preemptions are context switches forced by the scheduler (e.g., timeslice expiration)
- Non-preemptions – a thread voluntarily yields (e.g., blocking on an unavailable lock and thread end)

Thread 1

```
x = 1;
if (p != nullptr) {
  x = p->f;
}
```

Thread 2

```

p = nullptr;


```

Preemption bounding

- Systematically inserts a small number of preemptions
- Preemptions are context switches forced by the scheduler (e.g., timeslice expiration)
- Non-preemptions – a thread voluntarily yields (e.g., blocking on an unavailable lock and thread end)

Thread 1

```
x = 1;
if (p != nullptr) {

  ...

  x = p->f;
}
```

preempted

Thread 2

```



p = nullptr;
```

Preemption bounding

- Systematically inserts a small number of preemptions
- Preemptions are context switches forced by the scheduler (e.g., timeslice expiration)
- Non-preemptions – a thread voluntarily yields (e.g., blocking on an unavailable lock and thread end)

Thread 1

```
x = 1;
if (p != nullptr) {

  ...

  x = p->f;
}
```

preempted

Thread 2

```



p = nullptr;
```

Helps alleviate the problem of state space explosion

Advantages of preemption bounding

- Most errors are caused by few (<2) preemptions (similar to bug depth)
- Generates an easy to understand error trace
  - Preemption points almost always point to the root cause of the bug
- Leads to good heuristics
  - Insert more preemptions in code that needs to be tested
  - Avoid preemptions in libraries
  - Insert preemptions in recently modified code
- A good coverage guarantee to the user
  - When CHESS finishes exploration with 2 preemptions, any remaining bug requires 3 preemptions or more

Contributions of CHESS

Integrates stateless model checking ideas to testing concurrent programs with minimal perturbation

Ability to consistently reproduce erroneous interleavings

DTHREADS: Efficient and Deterministic Multithreading

Remember the Sources of Nondeterminism?

Sources of nondeterminism: input, environment, interleaving, other sources like compiler and hardware reordering

Deterministic Multithreading

- Deterministic execution can simplify multithreading
  - Executing the same program with same inputs will always provide same results
- Deterministic multithreading would simplify
  - Testing and debugging
  - Record and replay mechanism
  - Fault tolerance mechanisms

Different Interleavings are Possible

```c
int a = 0;
int b = 0;
int main() {
  pthread_create(&p1, NULL, thread1, NULL);
  pthread_create(&p2, NULL, thread2, NULL);
  pthread_join(&p1, NULL);
  pthread_join(&p2, NULL);
  printf("%d, %d\n", a, b);
}
```

```c
void* thread1(void*) {
  if (b == 0) {
    a = 1;
  }
  return NULL;
}

void* thread2(void*) {
  if (a == 0) {
    b = 1;
  }
  return NULL;
}
```

What are possible outputs?

Guarantees by DTHREADS

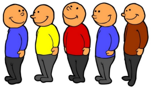
- DTHREADS guarantees deterministic execution of multithreaded programs even in the presence of data races
- Given the same sequence of inputs or OS events, a program using DTHREADS always produces the same output
- DTHREADS allows interleavings only at synchronization points
- DTHREADS uses synchronization operations as transactional boundaries
- Changing the code or input does not affect the schedule as long as the sequence of synchronization operations remains unchanged

T. Liu et al. DTHREADS: Efficient Deterministic Multithreading. SOSP, 2011.

How DTHREADS Provides Determinism

Isolation

Deterministic time



Deterministic order

T. Liu et al. DTHREADS: Efficient Deterministic Multithreading. SOSP, 2011.

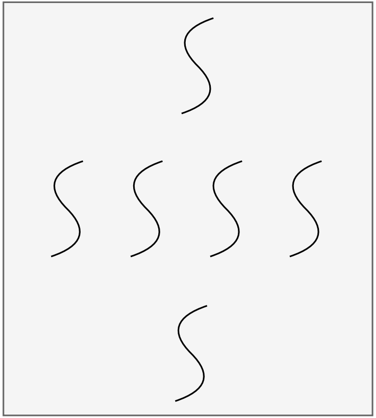
Deterministic Execution by DTHREADS

```
int a = 0;
int b = 0;
int main() {
  pthread_create(&p1, NULL, thread1, NULL);
  pthread_create(&p2, NULL, thread2, NULL);
  pthread_join(&p1, NULL);
  pthread_join(&p2, NULL);
  printf("%d, %d\n", a, b);
}
```

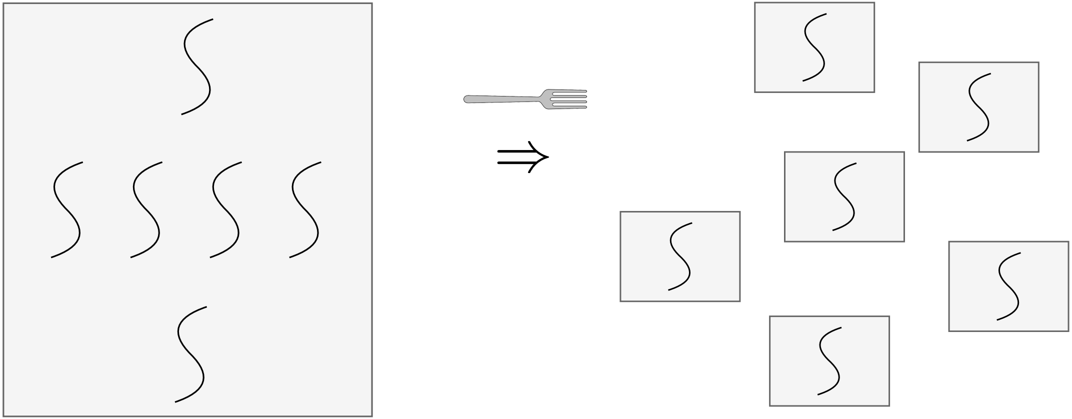
```
void* thread1(void*) {
  if (b == 0) {
    a = 1;
  }
  return NULL;
}

void* thread2(void*) {
  if (a == 0) {
    b = 1;
  }
  return NULL;
}
```

DTHREADS will always generate (1, 1) as the output

Shared Address Space



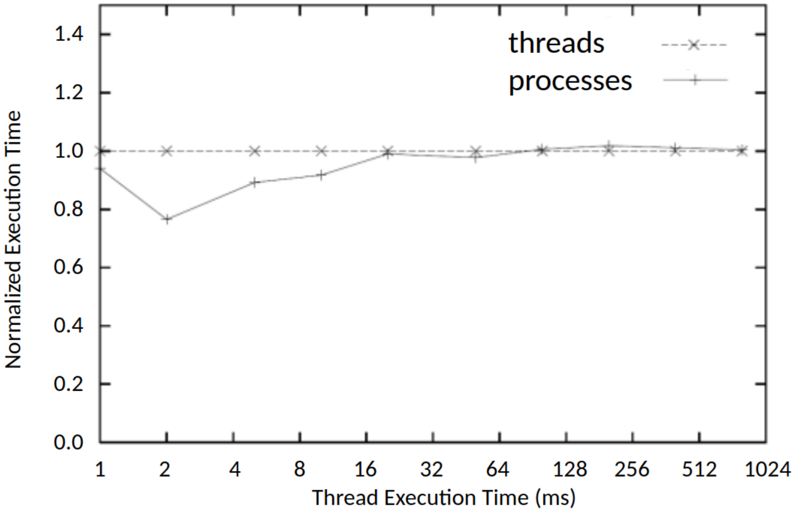
Shared vs Disjoint Address Space



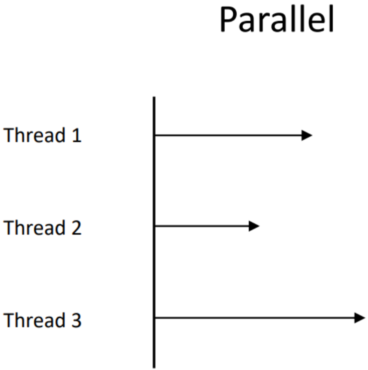
Isolated Memory Access

- Processes have separate address spaces, implies that updates to shared memory are not visible
- Updates are made visible only at synchronization points
- Code regions between synchronization operations behave as atomic transactions
- DTHREADS reimplements pthreads synchronization primitives to guarantee a deterministic ordering

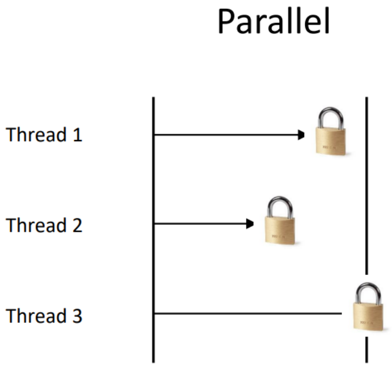
Performance of Threads vs Processes



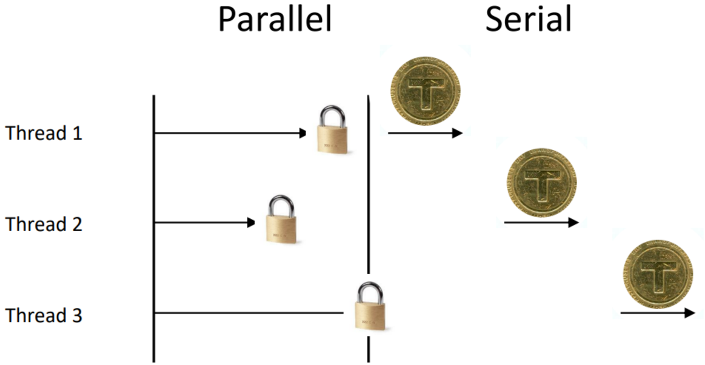
DTHREADS Phases



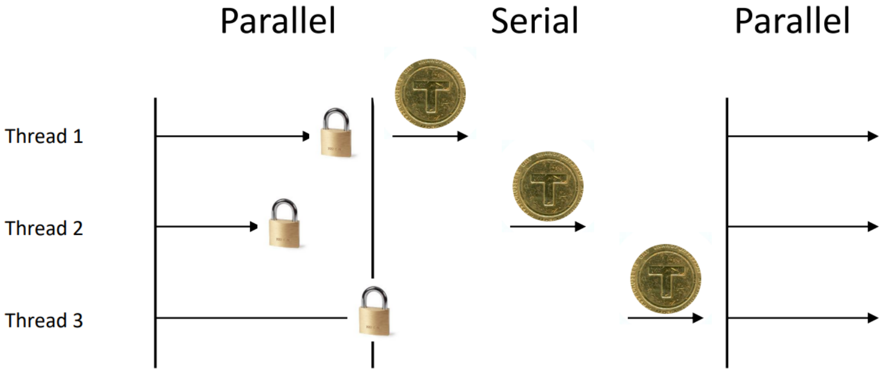
DTHREADS Phases



DTHREADS Phases



DTHREADS Phases



Shared-Memory Updates in Parallel Phase

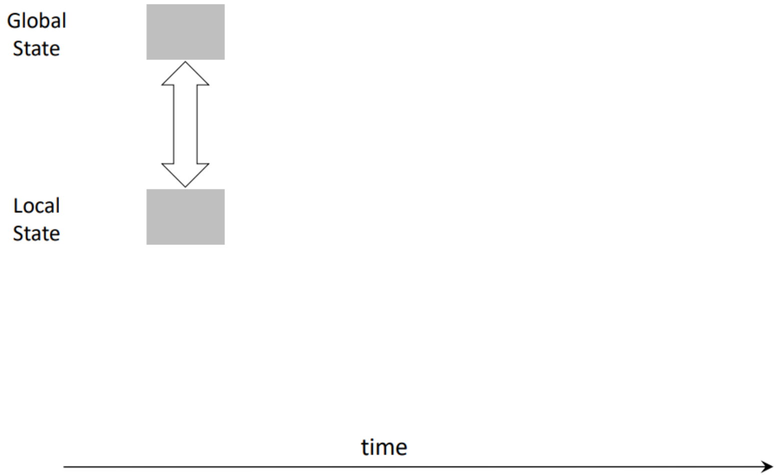
- DTHREADS uses memory-mapped files to share shared data (e.g., globals and heap) across processes
- Two copies of pages are created — one is read-only and the other is for local updates
- Threads have a read-only mapping of the shared pages at the beginning of the parallel phase
- Reads are performed from the shared page
- Upon a write, a private copy of the page is created (copy-on-write) and the write operates on the private copy

Snapshot pages before modifications

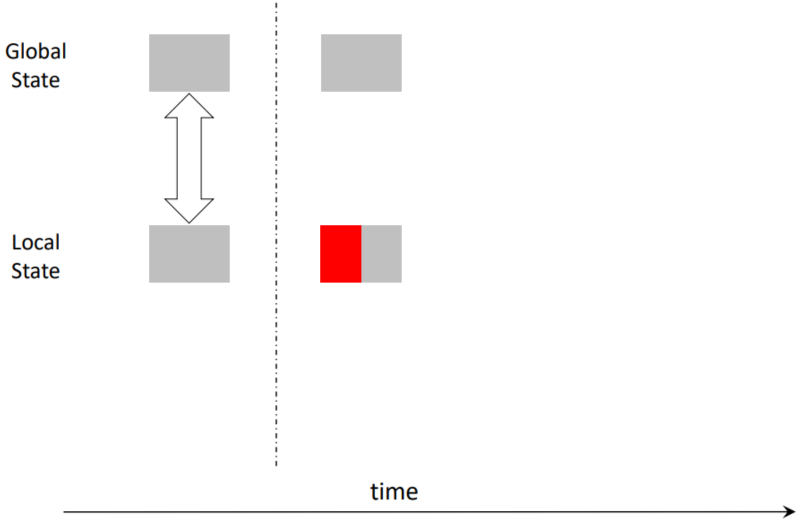
Snapshot pages before modifications

Write back diffs

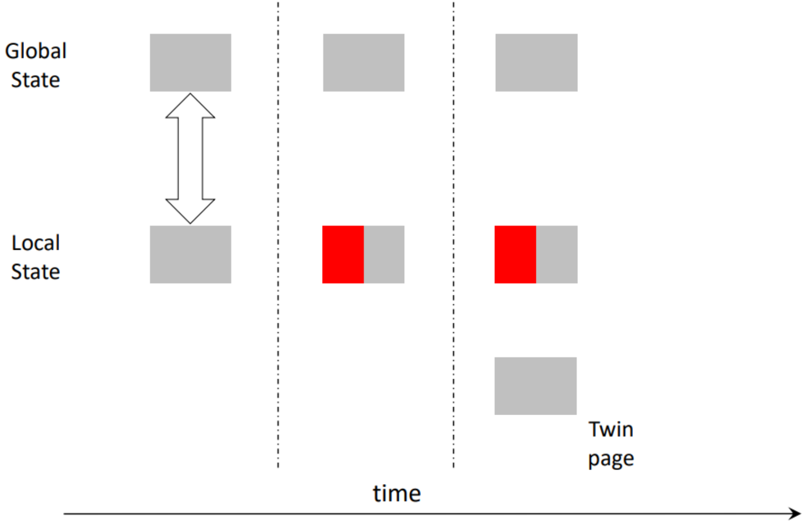
Commit Protocol



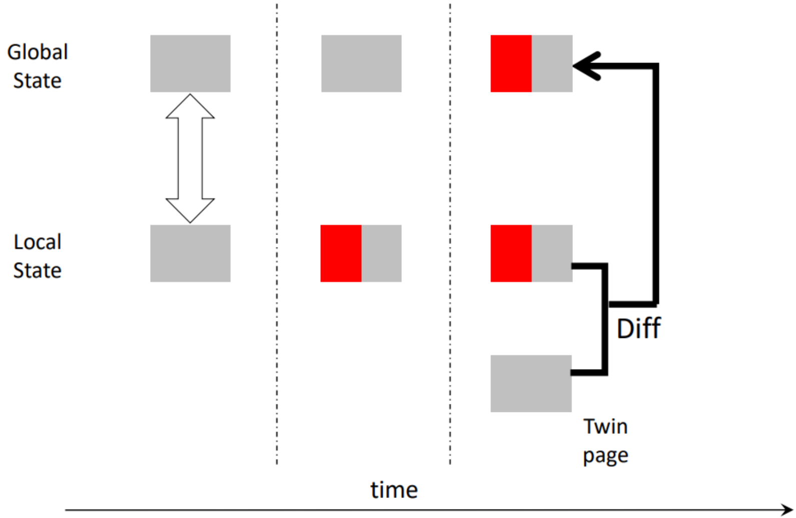
Commit Protocol



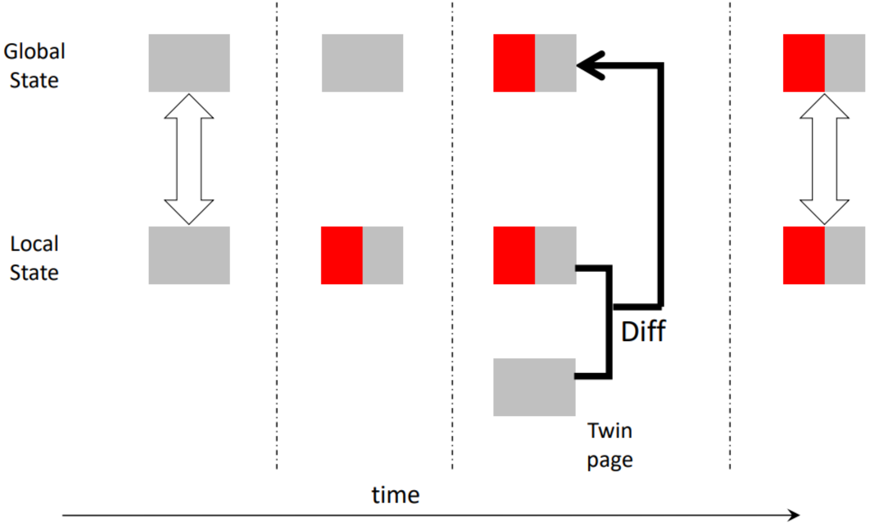
Commit Protocol



Commit Protocol



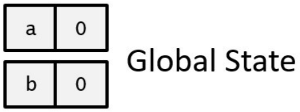
Commit Protocol



Commit Protocol

- During commit, DTHREADS compare the local copy with a “twin” copy of the original shared page
  - Writes back only the different bytes
  - First thread can copy back the whole page
- Private pages are released at the end of the serial phase

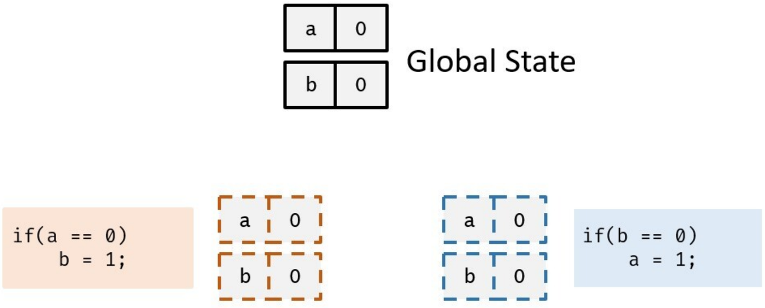
DTHREADS Example Execution



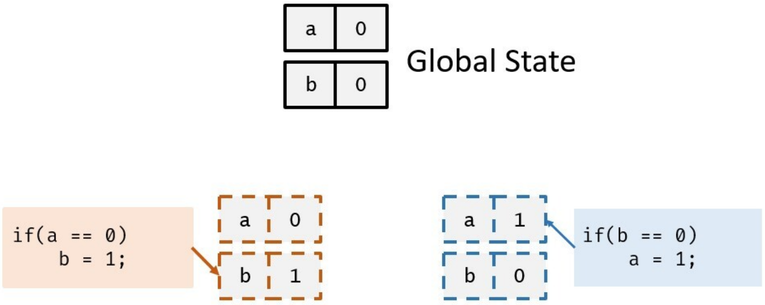

```
if(a == 0)
    b = 1;
```

```
if(b == 0)
    a = 1;
```

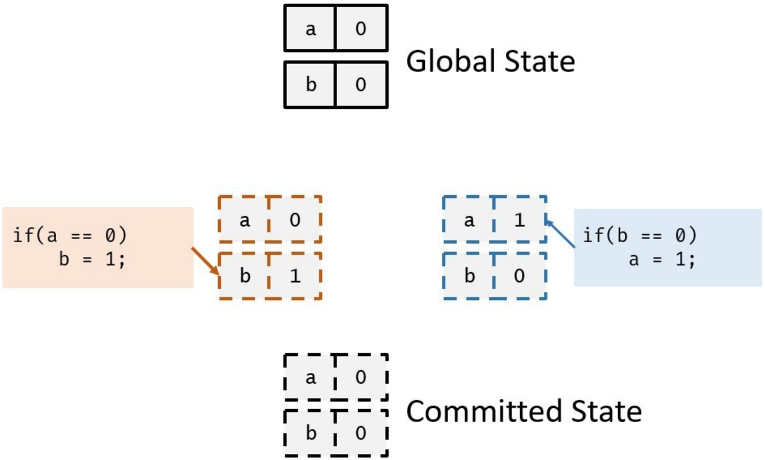
DTHREADS Example Execution



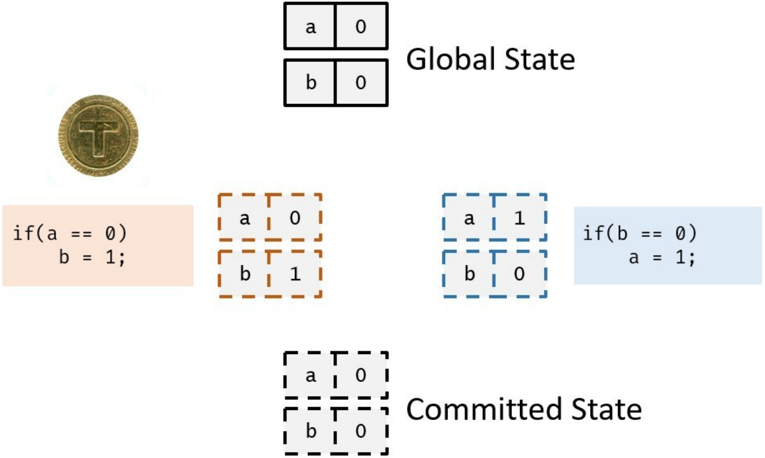
DTHREADS Example Execution



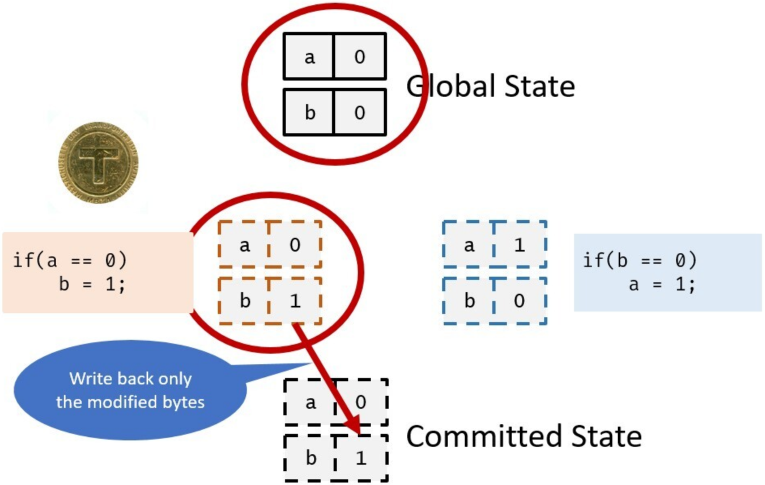
DTHREADS Example Execution



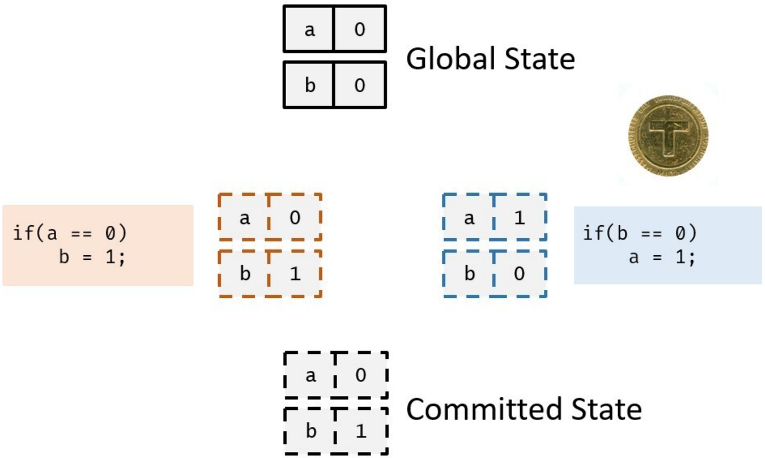
DTHREADS Example Execution



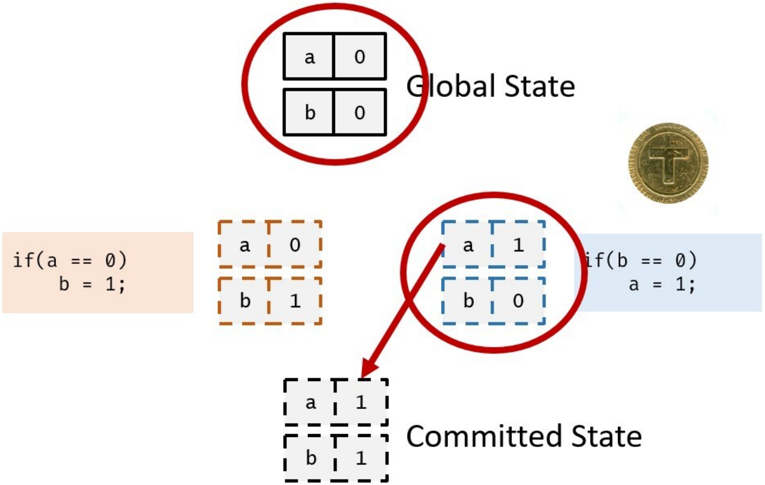
DTHREADS Example Execution



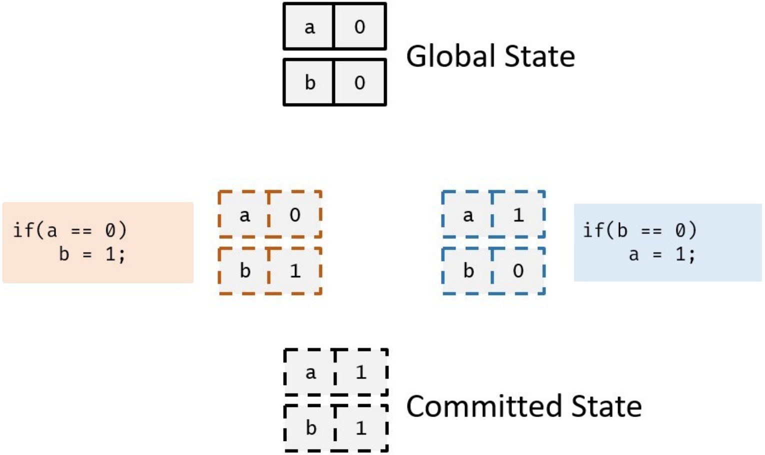
DTHREADS Example Execution



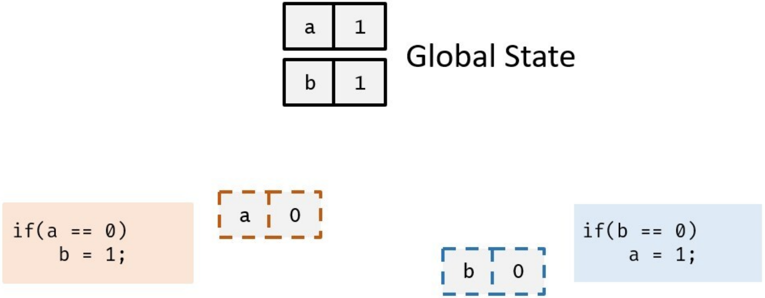
DTHREADS Example Execution

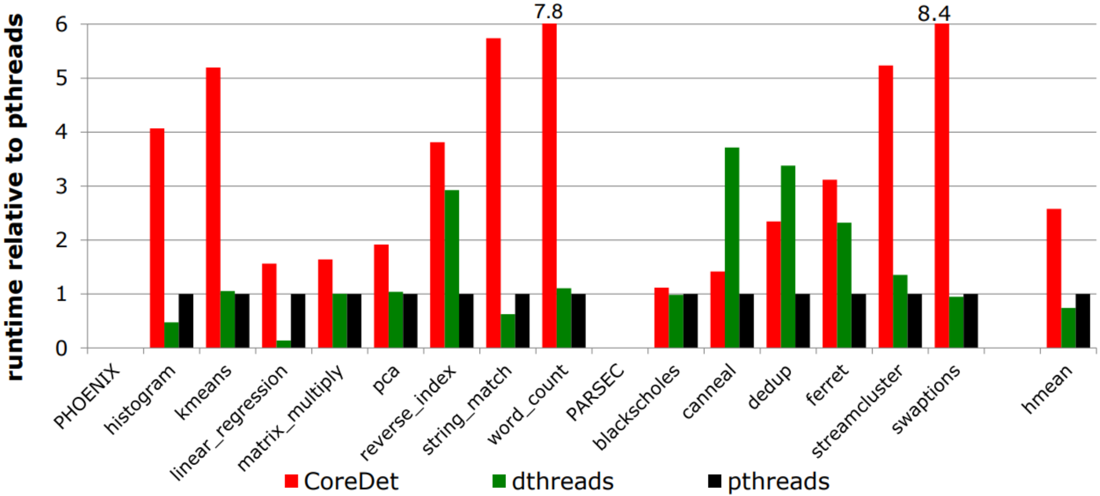


DTHREADS Example Execution



DTHREADS Example Execution





Generally as fast or faster than pthreads

Fuzzing Concurrent Programs

Fuzz Testing

Fuzzing is an automated software testing technique that is based on feeding the program with random inputs and monitoring the output

- Run the program with dynamic error detectors (e.g., Valgrind and AddressSanitizer)

Advantages

\+ Easy to set up, can treat the application as a blackbox

Disadvantages

– Probability of generating inputs that trigger an incorrect behavior is low if careful choices are not made

– Inputs often require structure, random inputs are likely to be malformed

AFL[†], AFL++[§], and libFuzzer[†] are popularly used fuzzers

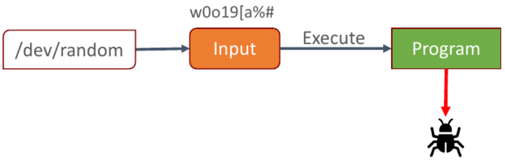
[†] american fuzzy lop

[§] American Fuzzy Lop plus plus (AFL++)

[†] libFuzzer — a library for coverage-guided fuzz testing

Origin of Fuzz Testing

- On a night in 1988, Barton Miller tried to connect to his Unix system in office via a dial up connection
- There was heavy rain and thunderstorm which introduced disturbances (i.e., “fuzz”)
- Crashed many UNIX utilities he had been using successfully everyday
- He realized that there was something fundamentally wrong with the applications
- Asked three groups in his seminar course to implement this idea of fuzz testing
  - Two groups failed to achieve any crash results!
  - The third group succeeded!
  - Crashed 25-33% of the utility programs on the seven Unix variants that they tested



1990 study found crashes in:
adb, as, bc, cb, col, diction, emacs, eqn, ftp, indent, lex, look, m4, make, nroff, plot, prolog, ptx, refer!, spell, style, tsort, uniq, vgrind, vi

B. Miller et al. An Empirical Study of the Reliability of UNIX Utilities. CACM, vol. 33, no. 12, pp. 32–44, Dec. 1990.

Types of Fuzz Testing

Blackbox

+ Generates test cases based on the specification
− Ignores implementation details, may miss testing boundary cases
  - May rerun the same path over again (i.e., low coverage)
  - May be very hard to generate inputs for certain paths with restrictive conditions
  - May cause the program to terminate for logical reasons — fail format checks and stop

Whitebox

- Fuzzing heuristics depend on the application internals to generate good test cases
- Tracks a coverage metric to estimate the quality of testing
- More smarter than blackbox, but complex and slower

Graybox

- Fuzzing based on code coverage
- Instrument the program to track coverage

Generating Inputs Randomly May Not be Effective

<APACHE ANT>

```
$ ant -f build.xml
```

```
<project default="dist">
 <target name="init">
  <mkdir dir="${build}"/>
 </target>
 …
```

```
$ ant -f /dev/random
```

```
lrha3wn5p0w3uz;54 p0a23
rw3i 50a20 5a2y58a2p
y3wry3p285
q@P"uer9zparu9apur9qa3802
y5o2y 392r523a90wesu
```

Generating Inputs Randomly May Not be Effective

<APACHE ANT>

`$ ant -f build.xml`

```
<project default="dist">
 <target name="init">
  <mkdir dir="${build}"/>
 </target>
 …
```

`$ ant -f /dev/random`

```
lrha3wn5p0w3uz;54 p0a23
rw3i 50a20 5a2y58a2p
y3wry3p285
q@P"uer9zparu9apur9qa3802
y5o2y 392r523a90wesu
```

```c
func(char *name, char *passwd, char *buf) {
  if (authenticate_user(name, passwd)) {
    if (check_format(buf)) {
      update(buf); // crash here
    }
  }
}
```

Mutation-based Fuzzing

- Take a well-formed input (i.e., seed) and randomly perturbs it (e.g., flip a bit) to generate new inputs
- Perturbation can use heuristics and domain knowledge

Binary input Flip bits or bytes and change random byte sequences

Text input Insert random symbols or keywords from a dictionary

\+ Little or no knowledge of the structure of the inputs and the application is required

– Still prone to problems

  - Dependent on the quality of the initial test corpus
  - May rerun the same path over again
  - May be very hard to generate inputs for certain paths with restrictive conditions

Generate Inputs Randomly via Mutation

```
$ ant -f build.xml
```

```
<project default="dist">
 <target name="init">
  <mkdir dir="${build}"/>
 </target>
 …
```

```
$ ant -f build.xml.mut
```

```
<project default="dist">
 <taWget name="init">
  <madir dir="2{build}"/@
 </tar?get>
 …
```

Mutation using Genetic Algorithms

- Mutational fuzzing can use genetic algorithms for generating mutations
- Genetic algorithms (GA) are search algorithms inspired from biology
  - Maintains a fixed-size population of possible solutions
  - Defines a set of mutation operators that combine solutions from the population to create new solutions
  - Applies the mutation operators to the current population to a create a new “generation” of solutions
  - Uses a fitness function (e.g., code coverage) to prune the set of possible solutions to keep the most promising ones
  - Repeats until some stopping criteria is met

Generational Fuzzing

- Test cases are generated from scratch
- Require some description of the input format: RFC and documentation
- Anomalies are added to each possible spot in the inputs

\+ Knowledge of protocol should give better results than random fuzzing

− Requires a specification for every input format

− Writing test case generators is non-trivial

Coverage-Guided Fuzzing

Idea: code that has not been covered by tests are likely to contain bugs

- Code coverage (e.g., line, branch, edge, or path) is used to determine how thoroughly code has been tested
- Steps in coverage-based fuzzing
  - Start with an initial user-provided test suite T
  - Observe and track coverage while running tests from T
  - Mutate test cases in T to generate new tests $T^{'}$
  - Run new tests from $T^{'}$
  - Move those tests that lead to new coverage from $T^{'}$ to T
  - Continue fuzzing until the coverage goal is met
- Effectiveness of fuzzing is determined by the coverage of the program by the test suite
- Such an objective metric has many uses: stop testing, compare the quality of test suites, and generate test cases

Graybox Fuzzing Workflow

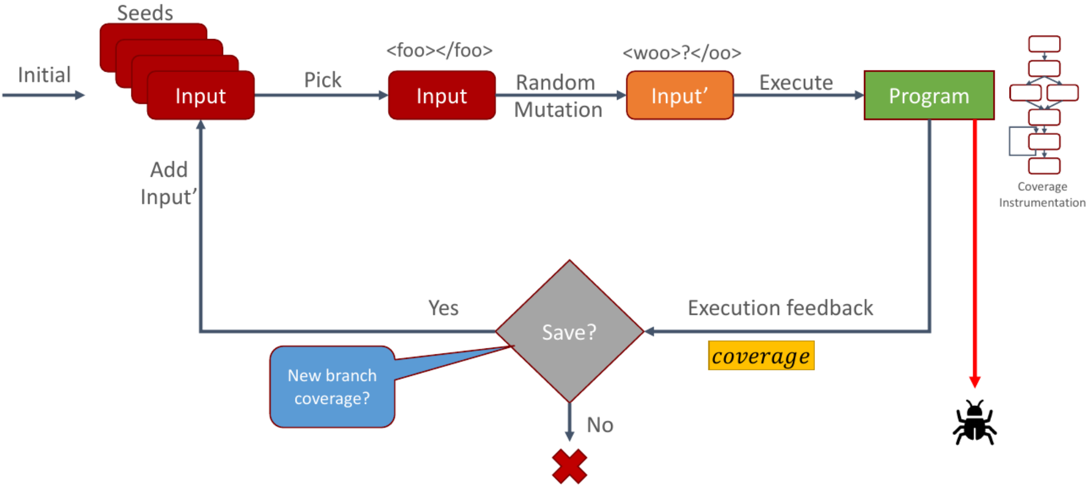
Input program P_o, initial seed queue Q_S

Output final seed queue Q_S, vulnerable seed files T_C

Steps

```
P_f ← instrument(P_o)                                  ▷ instrumentation
T_C ← Φ
while true do
    t ← select_next_seed(Q_S)                          ▷ seed selection
    M ← get_mutation_chance(P_f, t)                    ▷ seed scheduling
    for i ∈ {1, ..., M} do
        t' ← mutated_input(t)                          ▷ seed mutation
        res ← execute(P_f, t', N_c)                    ▷ repeated execution
        if is_interesting(res) then                    ▷ seed triaging
            T_C ← T_C ∪ {t'}                           ▷ report
        else if new_coverage(t', res) then
            Q_S ← Q_S ⊕ t'                             ▷ preserve effective seeds
```

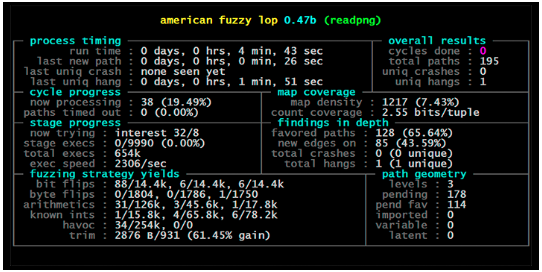
Coverage-Guided Fuzzing



https://cmu-program-analysis.github.io/2022/lecture-slides/20-fuzzing.pdf

Coverage-Guided Fuzzing with AFL

- One of the first popular coverage-guided fuzzers
  - Started by Michal Zalewski (lcamtuf)
- AFL instruments branch statements and tracks code paths taken at run time
- AFL is very easy to use and has been very effective
  - Provides a GCC wrapper to instrument the code
  - Uses counters to track edges in the control flow graph
  - Uses hashing to encode different edges (imprecise but efficient)



http://lcamtuf.coredump.cx/afl/

Comparing Fuzzing Approaches

- Graybox fuzzing (e.g., AFL, libFuzzer, and HonggFuzz)
  - \+ Requires minimal setup similar to blackbox fuzzing
  - \+ More targeted than blackbox fuzzing, but does not understand the program
  - − Searches for inputs independently from the program
  - − May not be able to execute some code paths

- Whitebox fuzzing
  - Couples test case generation with fuzzing
  - Test generation is based on static analysis and/or symbolic execution
    - Run the code with some initial input
    - Collect constraints on input with symbolic execution
    - Generate new constraints
    - Solve constraints with constraint solver
    - Synthesize new inputs
  - Rather than generating new inputs and checking whether they cover a new path, compute inputs that **will execute a desired** path

Challenges with Fuzzing

- Mutation heuristics
  - Which inputs to mutate? How many times? How to generate meaningful test cases?
- Coverage
  - What to instrument to improve feedback? How to keep overhead low?
- Oracle
  - How to monitor the application to find a bug?
    - For example, a crash or silent overflow or infinite loop or race conditions?
  - Instrument the program with runtime sanitizers to monitor abnormal program execution
  - Use Valgrind or sanitizers† (e.g., ASAN, TSAN, and UBSAN)
- When do we stop fuzzing?
  - Need to balance cost vs bug coverage

†https://github.com/google/sanitizers

Power Schedules with Mutational Fuzzing

- Consider a new generation of test inputs containing
  - $n - 1$ inputs that have been in the population for at least a few generations,
  - one input that covered a new branch or path that was created in the last round of mutation
- Which input should we mutate?
  - Intuitively, we expect that the new input should be mutated more often in the next generation
  - This intuition is implemented via power schedules

Martin Kellogg. CS 684: Testing and Quality Assurance: Fuzzing.

Power Schedules with Mutational Fuzzing

- A power schedule distributes fuzzing time among the seeds in the population
- Each seed is assigned an energy value using a policy
  - Seeds that exercise rarely-covered paths have more energy
  - Seeds that exercise code close to the area of interest (e.g., modifications) is given more energy (called directed fuzzing)
- The chances of mutating a seed are proportional to its energy
- Usual policy is:
  - Newly-discovered seeds start with high energy
  - When a seed is mutated to produce an input that increases fitness, its energy increases
  - When a seed is mutated but does not produce an input that increases fitness, its energy decreases

Martin Kellogg. CS 684: Testing and Quality Assurance: Fuzzing.

Fuzzing Concurrent Programs

- Goal is to use fuzzing to detect concurrency bugs like data races and deadlocks
  (i) Explore as many code paths and thread interleavings as possible
  (ii) Use a "good" bug detection algorithm

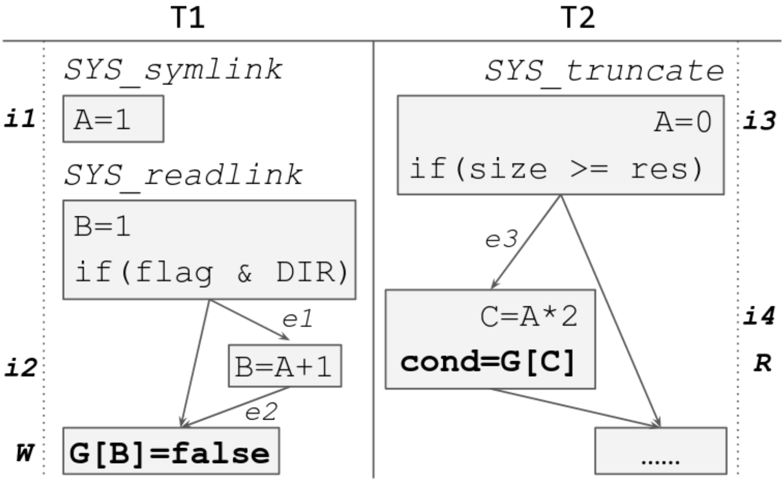
- How about reusing existing pipelines meant for sequential programs?
  - For example, AFL+TSAN or Syzkaller+KCSAN for data races

— Existing fuzzers use coverage meant for sequential programs (e.g., branch coverage)

— Do not effectively prioritize exploring thread interleavings

https://github.com/google/syzkaller

Limitations with Branch Coverage



Limitations with Branch Coverage

T1	T2
A=1	
B=A+1	
	A=0
	C=A*2

① B=2, C=0 *<nil>*

T1	T2
A=1	
	A=0
B=A+1	
	C=A*2

② B=1, C=0 *i3→i2*

T1	T2
A=1	
	A=0
	C=A*2
B=A+1	

③ B=1, C=0 *i3→i2*

T1	T2
	A=0
	C=A*2
A=1	
B=A+1	

④ B=2, C=0 *<nil>*

T1	T2
	A=0
A=1	
	C=A*2
B=A+1	

⑤ B=2, C=2 *i1→i4*

T1	T2
	A=0
A=1	
B=A+1	
	C=A*2

⑥ B=2, C=2 *i1→i4*

Concurrency Coverage

- Check for bugs among possibly overlapping concurrent instructions from different threads
- Alias instruction pair describes the locations of two concurrently-executed instructions
- Alias coverage tracks how many such interleaving points have been covered during testing

M. Xu et al. KRACE: Data Race Fuzzing for Kernel File Systems. S&P, 2020.

Data Race from JFS (Linux kernel v5.4)

Thread 1

File: `linux/fs/jfs/jfs_txnmgr.c`

```c
void txEnd(...) {
  ...
  // racy read
  log = JFS_SBI(tblk->sb)->log;
  ...
  if (--log->active == 0)
  ...
}
```

Thread 2

File: `linux/fs/jfs/jfs_logmgr.c`

```c
int lmLogClose(...) {
  ...
  struct jfs_sb_info *sbi = JFS_SBI(sb);
  ...
  // racy write
  sbi->log = NULL;
  ...
}
```

The data race was introduced in Linux kernel 2.6.12 in June 2005 and was hidden for fifteen years

Importance of Context-Sensitive Call Pairs

Call Pair 1

Thread 1	`jfs_lazycommit() -> txLazyCommit() -> txEnd()`
Thread 2	`jfs_put_super() -> jfs_umount() -> lmLogClose()`

Call Pair 2

Thread 1	`jfs_lazycommit() -> txLazyCommit() -> txEnd()`
Thread 2	`jfs_remount() -> jfs_umount() -> lmLogClose()`

Context-Sensitive Concurrency Coverage

Maintain information of a function call (*CallInfo*) as a tuple of the call site (*CallLoc*) and the location of the function definition (*FuncLoc*)

$$CallInfo = [CallLoc, FuncLoc]$$

Maintain the calling context (*CallCtx*) as the list of function calls in the run-time call stack

$$CallCtx = [CallInfo_1, CallInfo_2]$$

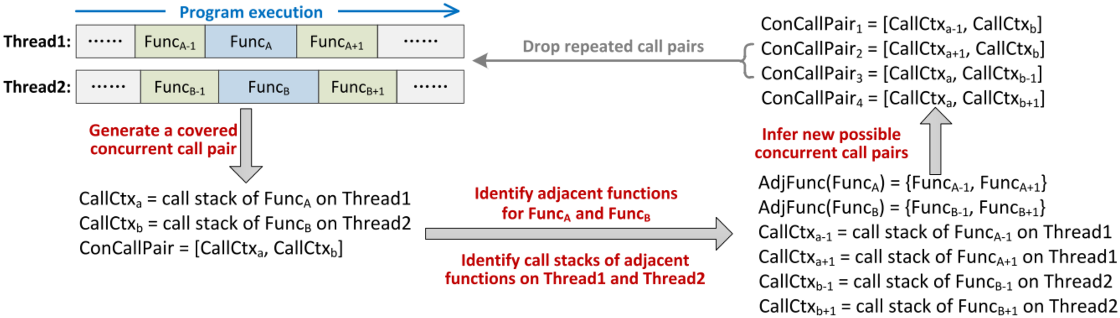
Concurrent call pair maintains the calling contexts of concurrently executing functions

$$ConcCallPair = \{CallCtx_1, CallCtx_2\}$$

Z. Jiang et al. Concurrency Fuzzing for Data-Race Detection. NDSS, 2022.

Adjacency-Directed Mutation

If two functions are concurrently executed, the adjacent functions in their call stacks can probably be executed concurrently as well



Z. Jiang et al. Concurrency Fuzzing for Data-Race Detection. NDSS, 2022.

References I

- D. Hovemeyer and W. Pugh. Finding Concurrency Bugs in Java. PODC Workshop on Concurrency and Synchronization in Java Programs, 2004.
- D. Hovemeyer and W. Pugh. Finding Bugs is Easy. OOPSLA, 2004.
- S. Burckhardt et al. A Randomized Scheduler with Probabilistic Guarantees of Finding Bugs. ASPLOS, 2010.
- M. Musuvathi et al. Finding and Reproducing Heisenbugs in Concurrent Programs. OSDI 2008.
- S. Nagarakatte et al. Multicore Acceleration of Priority-Based Schedulers for Concurrency Bug Detection. PLDI, 2012.
- S. Burckhardt et al. CHESS: Analysis and Testing of Concurrent Programs. Tutorial at PLDI, 2009.
- M. Musuvathi. Randomized Algorithms for Concurrency Testing. CONCUR, 2017.
- T. Liu et al. DTHREADS: Efficient Deterministic Multithreading. SOSP, 2011.

References II

- H. Chen et al. MUZZ: Thread-aware Grey-box Fuzzing for Effective Bug Hunting in Multithreaded Programs. Usenix Security, 2020.
- M. Xu et al. KRACE: Data Race Fuzzing for Kernel File Systems. S&P, 2020.
- Z. Jiang et al. Concurrency Fuzzing for Data-Race Detection. NDSS, 2022.
- D. Wolff et al. Greybox Fuzzing for Concurrency Testing. ASPLOS, 2024.