

# CS 5/6110, Software Correctness Analysis, Spring 2022

Ganesh Gopalakrishnan  
School of Computing  
University of Utah  
**Salt Lake City, UT 84112**



# Lecture n : Dynamic Symbolic Execution (also known as “Concolic Execution” standing for Concrete + Symbolic Execution)

- There is so much C codes out there
- How to even make a dent in verifying them?
- How do DSE or Concolic tools help?
  - An amazing success story in this area: KLEE – OSDI 2008
- How do they compare with fuzzers (what are they?)
  - Mention about American Fuzzy Lop or AFL
  - Study by Dr. Peng Li (my former student) is very good – slides uploaded on Canvas
- How does all of this apply to parallel computing
  - Say GPUs?
  - Test of Time Award Honorable Mention for Dr. Guodong Li (my former student) and me at FSE 2020
    - Work on PUG (2010) and work on GKLEE (2012) - Guodong was VERY foresighted to have keyed off KLEE in 4 years!
- Dr. Guodong Li is with Google Research now, and Dr. Peng Li is with ByteDance Inc. now

# How hard is it?

# **KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs**

Cristian Cadar, Daniel Dunbar, Dawson Engler \*  
*Stanford University*

```

1 : void expand(char *arg, unsigned char *buffer) { 8
2 :     int i, ac; 9
3 :     while (*arg) { 10*
4 :         if (*arg == '\\') { 11*
5 :             arg++;
6 :             i = ac = 0;
7 :             if (*arg >= '0' && *arg <= '7') {
8 :                 do {
9 :                     ac = (ac << 3) + *arg++ - '0';
10:                    i++;
11:                } while (i<4 && *arg>='0' && *arg<='7');
12:                *buffer++ = ac;
13:            } else if (*arg != '\\0')
14:                *buffer++ = *arg++;
15:            } else if (*arg == '[') { 12*
16:                arg++;
17:                i = *arg++;
18:                if (*arg++ != '-' ) { 13
19:                    *buffer++ = '[';
20:                    arg -= 2;
21:                    continue;
22:                }
23:                ac = *arg++;
24:                while (i <= ac) *buffer++ = i++; 14
25:                arg++; /* Skip ']' */
26:            } else { 15!
27:                *buffer++ = *arg++;
28:            }
29:        }
30:    ...
31:    int main(int argc, char* argv[]) { 1
32:        int index = 1; 2
33:        if (argc > 1 && argv[index][0] == '-') { 3*
34:            ...
35:        } 4
36:        ...
37:        expand(argv[index++], index); 5
38:        ... 6
39:    } 7

```

**Figure 1:** Code snippet from MINIX’s `tr`, representative of the programs checked in this paper: tricky, non-obvious, difficult to verify by inspection or testing. The order of the statements on the path to the error at line 18 are numbered on the right hand side.

1 KLEE constructs symbolic command line string arguments whose contents have no constraints other than zero-termination. It then constrains the number of arguments to be between 0 and 3, and their sizes to be 1, 10 and 10 respectively. It then calls `main` with these initial path constraints.

2 When KLEE hits the branch `argc > 1` at line 33, it uses its constraint solver STP [23] to see which directions can execute given the current path condition. For this branch, both directions are possible; KLEE forks execution and follows both paths, adding the constraint `argc > 1` on the false path and `argc ≤ 1` on the true path.

3 Given more than one active path, KLEE must pick which one to execute first. We describe its algorithm in Section 3.4. For now assume it follows the path that reaches the bug. As it does so, KLEE adds further constraints to the contents of `arg`, and forks for a total of five times (lines denoted with a “\*”): twice on line 33, and then on lines 3, 4, and 15 in `expand`.

4 At each dangerous operation (e.g., pointer dereference), KLEE checks if any possible value allowed by the current path condition would cause an error. On the annotated path, KLEE detects no errors before line 18. At that point, however, it determines that input values exist that allow the read of `arg` to go out of bounds: after taking the true branch at line 15, the code increments `arg` twice without checking if the string has ended. If it has, this increment skips the terminating '`\0`' and points to invalid memory.

5 KLEE generates concrete values for `argc` and `argv` (i.e., `tr [ " " "`) that when rerun on a raw version of `tr` will hit this bug. It then continues following the current path, adding the constraint that the error does not occur (in order to find other errors).

# Why can't we crank through all inputs/states?

Your answer here for a C program that has exactly one 32-bit register

How about a 64-bit register?

Your answer here for a C program with 1k bytes of state and 64 bits of input

How about a C program with 5 threads with 5 sequential steps? How many interleavings?

How about something larger?

# The main technology!

- Progress in Boolean Satisfiability is central to the magic I'm going to demo
  - SAT-solving increased by 1000x or more (in performance)
- SAT alone is not sufficient
  - SMT-solving was essential
    - Think of the SEND + MORE = MONEY problem!
- Progress in SMT-solving happened this way
  - Nelson/Oppen methods of the 1980's and Shostak's method
  - SVC and CVC and Yices
  - Then Z3, and now CVC4 and a whole array of others!
- Rides on the power of SAT-solving but their “theory-solving” is key
- KLEE uses “STP”, an SMT solver that can handle bit-arrays and uninterpreted functions well
- This is why we must study mathematical logic
  - took 1000s of years since Euclid, 170 years since Boole, 100 years since Shannon, and 120 years since Russell, Peano and others to get here!
  - in addition to all those other “cool topics” to which students are flocking these days!
    - (What are some of those?!)

Demo-1  
: a hard-to-test program under conventional testing and then KLEE

```
/*
 * An error that is hard to hit
 */

#include <stdio.h>
#include <assert.h>
#include <stdlib.h>
#include "klee/klee.h"

int rare(int x) {
    if (x < RAND_MAX/2) {
        if (x > RAND_MAX/2 - 3) {
            assert(0);
        }
    } else
        return(1);
}

int main() {
    int a, i;

#ifdef KLEEON
    klee_make_symbolic(&a, sizeof(a), "a");
    rare(a);
#endif

#ifdef PRINTON
#ifndef KLEEON
    printf("randmax = %d\n", RAND_MAX);
    for (i=0; i < 10000000; i++) {
        a = rand(); //% 100000000;
        rare(a);
    }
#endif
#endif
}
```

Demo-1  
: a hard-  
to-test  
program  
under  
conven-  
tional  
testing  
and  
then  
KLEE

```
/*
 * An error that is hard to hit
 */

#include <stdio.h>
#include <assert.h>
#include <stdlib.h>
#include "klee/klee.h"

int rare(int x) {
    if (x < RAND_MAX/2) {
        if (x > RAND_MAX/2 - 3) {
            assert(0);
        }
    } else
        return(1);
}

int main() {
    int a, i;

#ifndef KLEEON
    klee_make_symbolic(&a, sizeof(a), "a");
    rare(a);
#endif

#ifndef PRINTON
#ifndef KLEEON
    printf("randmax = %d\n", RAND_MAX);
    for (i=0; i < 10000000; i++) {
        a = rand(); /*% 10000000;
        rare(a);
    }
#endif
#endif
}
```

Enter

KLEE !!

```
/*
 * An error that is hard to hit
 */

#include <stdio.h>
#include <cassert.h>
#include <stdlib.h>
#include "klee/klee.h"

int rare(int x) {
    if (x < RAND_MAX/2) {
        if (x > RAND_MAX/2 - 3) {
            assert(0);
        }
    } else
        return(1);
}

int main() {
    int a, i;

#ifdef KLEEON
    klee_make_symbolic(&a, sizeof(a), "a");
    rare(a);
#endif

#ifdef PRINTON
#ifndef KLEEON
    printf("randmax = %d\n", RAND_MAX);
    for (i=0; i < 10000000; i++) {
        a = rand(); /* 100000000;
        rare(a);
    }
#endif
#endif
}
```

```
klee@f88364af576a:~/klee_src/examples/hardToHit$ clang -DKLEEON -I ../../include -emit-llvm -g -c -O0 -Xclang -disable-O0-optnone hardToHit.c
hardToHit.c:18:1: warning: control may reach end of non-void function [-Wreturn-type]
}
^
1 warning generated.
klee@f88364af576a:~/klee_src/examples/hardToHit$ klee hardToHit.bc
KLEE: output directory is "/home/klee/klee_src/examples/hardToHit/klee-out-3"
KLEE: Using STP solver backend
[KLEE: ERROR: hardToHit.c:13: ASSERTION FAIL: 0
[KLEE: NOTE: now ignoring this error at this location

KLEE: done: total instructions = 27
KLEE: done: completed paths = 3
KLEE: done: generated tests = 3
klee@f88364af576a:~/klee_src/examples/hardToHit$ echo "see we already hit the assert in concolic"
see we already hit the assert in concolic
klee@f88364af576a:~/klee_src/examples/hardToHit$ echo "now to see the tests synthesized"
[now to see the tests synthesized
klee@f88364af576a:~/klee_src/examples/hardToHit$ ls -ld klee-last
[lrwxrwxrwx 1 klee klee 49 Feb 10 18:34 klee-last -> /home/klee/klee_src/examples/hardToHit/klee-out-3
klee@f88364af576a:~/klee_src/examples/hardToHit$ echo "this directory has the tests"
>this directory has the tests
klee@f88364af576a:~/klee_src/examples/hardToHit$ ktest-tool klee-last/test000002.ktest
ktest file : 'klee-last/test000002.ktest'
args       : ['hardToHit.bc']
num objects: 1
object 0: name: 'a'
object 0: size: 4
object 0: data: b'\x00\x00\x00\x00'
object 0: hex : 0x00000000
object 0: int : 0
object 0: uint: 0
object 0: text: ....
klee@f88364af576a:~/klee_src/examples/hardToHit$ ktest-tool klee-last/test000001.ktest
ktest file : 'klee-last/test000001.ktest'
args       : ['hardToHit.bc']
num objects: 1
object 0: name: 'a'
object 0: size: 4
object 0: data: b'\xff\xff\xff?'
object 0: hex : 0xfffffff3
object 0: int : 1073741821
object 0: uint: 1073741821
object 0: text: ...?
klee@f88364af576a:~/klee_src/examples/hardToHit$ ls klee-last
assembly.ll messages.txt run.stats          test000001.kquery test000002.ktest warnings.txt
info      run.istats  test000001.assert.err test000001.ktest test000003.ktest
klee@f88364af576a:~/klee_src/examples/hardToHit$ echo "ah ha , test 000001 is the one that hit assert"
[ah ha , test 000001 is the one that hit assert
```

```
/*
 * An error that is hard to hit
 */

#include <stdio.h>
#include <cassert.h>
#include <stdlib.h>
#include "klee/klee.h"

int rare(int x) {
    if (x < RAND_MAX/2) {
        if (x > RAND_MAX/2 - 3) {
            assert(0);
        }
    } else
        return(1);
}

int main() {
    int a, i;

#ifdef KLEEON
    klee_make_symbolic(&a, sizeof(a), "a");
    rare(a);
#endif

#ifdef PRINTON
#ifndef KLEEON
    printf("randmax = %d\n", RAND_MAX);
    for (i=0; i < 10000000; i++) {
        a = rand(); /* 100000000;
        rare(a);
    }
#endif
#endif
}
```

Enter

KLEE !!

```
klee@f88364af576a:~/klee_src/examples/hardToHit$ gcc -DKLEEON -DPRINTON -I ../../include -L /home/klee/klee_build/lib hardToHit.c -lkleeRuntst
klee@f88364af576a:~/klee_src/examples/hardToHit$ KTEST_FILE=klee-last/test00002.ktest ./a.out
klee@f88364af576a:~/klee_src/examples/hardToHit$ KTEST_FILE=klee-last/test00001.ktest ./a.out
[a.out: hardToHit.c:13: rare: Assertion `0' failed.
Aborted
klee@f88364af576a:~/klee_src/examples/hardToHit$ echo "BOOM - we hit the assert "
BOOM - we hit the assert
klee@f88364af576a:~/klee_src/examples/hardToHit$
```

Demo-2  
Binary  
Search

Looks  
OK?

Ship it?!

```
#include <klee/klee.h>
#include <stdio.h>
#include <assert.h>
#include <stdlib.h>

void print_data(int arr[], int size, int target) {
    printf("searching for %d in:\n", target);
    for (int i=0; i < size-1; i++) {
        printf("%d, ", arr[i]);
    }
    printf("%d]\n", arr[size-1]);
}

int binary_search(int arr[], int size, int target) {
    #ifdef PRINTON
        print_data(arr, size, target);
    #endif
    int low = 0;
    int high = size - 1;
    int mid;
    while (low <= high) {
        mid = (low + high)/2;
        if (arr[mid] == target) {
            return mid;
        }
        if (arr[mid] < target) {
            low = mid + 1;
        }
        if (arr[mid] > target) {
            high = mid - 1;
        }
    }
    return -1;
}
```

```
int main() {

    int a[4]; // was a[10]

    #ifdef KLEEON
        klee_make_symbolic(&a, sizeof(a), "a");
        klee_assume(a[0] <= a[1]);
        klee_assume(a[1] <= a[2]);
        klee_assume(a[2] <= a[3]);

        klee_make_symbolic(&x, sizeof(x), "x");
    #endif

    #ifdef INITON
        a[0]=rand()%30;
        a[1]=a[0]+rand()%30;
        a[2]=a[1]+rand()%30;
        a[3]=a[2]+rand()%30;
    #endif

    int result = binary_search(a, 4, x); // was 10

    #ifdef PRINTON
        printf("result = %d\n", result);
    #endif
    // check correctness

    if (result != -1) {
        assert(a[result] == x);
    } else {
        // if result == -1, then we didn't find it.
        for (int i = 0; i < 3; i++) { // was 10
            assert(a[i] != x);
        }
    }
    return 1;
}
```

## Demo-2 Binary Search

Looks  
OK?

```
#include <klee/klee.h>
#include <stdio.h>
#include <assert.h>
#include <stdlib.h>

void print_data(int arr[], int size, int target) {
    printf("searching for %d in:\n", target);
    for (int i=0; i < size-1; i++) {
        printf("%d, ", arr[i]);
    }
    printf("%d\n", arr[size-1]);
}

int binary_search(int arr[], int size, int target) {
    #ifdef PRINTON
    print_data(arr, size, target);
    #endif
    int low = 0;
    int high = size - 1;
    int mid;
    while (low <= high) {
        mid = (low + high)/2;
        if (arr[mid] == target) {
            return mid;
        }
        if (arr[mid] < target) {
            low = mid + 1;
        }
        if (arr[mid] > target) {
            high = mid - 1;
        }
    }
    return -1;
}

int main() {
    int a[4]; // was a[10]
    #ifdef KLEEON
    klee_make_symbolic(&a, sizeof(a), "a");
    #endif
}
```

```
klee@f88364af576a:~/klee_src/examples/binsrch$ gcc -DINITON -DPRINTON -I ../../include binsrch.c
```

```
klee@f88364af576a:~/klee_src/examples/binsrch$ ./a.out
```

```
searching for -336381299 in:
```

```
[13, 29, 56, 81]
```

```
result = -1
```

```
klee@f88364af576a:~/klee_src/examples/binsrch$ clang -DKLEEON -I ../../include -emit-llvm -g -c -O0 -Xclang -disable-O0-optnone binsrch.c
```

```
klee@f88364af576a:~/klee_src/examples/binsrch$ klee binsrch.bc
```

```
KLEE: output directory is "/home/klee/klee_src/examples/binsrch/klee-out-18"
```

```
KLEE: Using STP solver backend
```

```
KLEE: WARNING: undefined reference to function: printf
```

```
KLEE: done: total instructions = 638
```

```
KLEE: done: completed paths = 9
```

```
KLEE: done: generated tests = 9
```

## Demo-2 Binary Search

# Looks OK?

# Ship it?!

```
#include <klee/klee.h>
#include <stdio.h>
#include <assert.h>
#include <stdlib.h>

void print_data(int arr[], int size, int target) {
    printf("searching for %d in:\n[%d, %d]", target);
    for (int i=0; i < size-1; i++) {
        printf("%d, ", arr[i]);
    }
    printf("%d]\n", arr[size-1]);
}

int binary_search(int arr[], int size, int target) {
#ifndef PRINTON
    print_data(arr, size, target);
#endif
    int low = 0;
    int high = size - 1;
    int mid;
    while (low <= high) {
        mid = (low + high)/2;
        if (arr[mid] == target) {
            return mid;
        }
        if (arr[mid] < target) {
            low = mid + 1;
        }
        if (arr[mid] > target) {
            high = mid - 1;
        }
    }
    return -1;
}

int main() {
    int a[4]; // was a[10]

#ifndef KLEEEON
    Klee_make_symbolic(&a, sizeof(a), "a");
    Klee_assume(a[0] <= a[1]);
    Klee_assume(a[1] <= a[2]);
    Klee_assume(a[2] <= a[3]);
    Klee_make_symbolic(&x, sizeof(x), "x");
#endif

#ifndef INITON
    a[0]=rand()%30;
    a[1]=a[0]+rand()%30;
    a[2]=a[1]+rand()%30;
    a[3]=a[2]+rand()%30;
#endif

    int result = binary_search(a, 4, x); // was 10

#ifndef PRINTON
    printf("result = %d\n", result);
#endif
    // check correctness

    if (result != -1) {
        assert(a[result] == x);
    } else {
        // if result == -1, then we didn't find it.
    }
    for (int i = 0; i < 3; i++) { // was 10

        assert(a[i] != x);
    }
}
return 1;
}
```

Demo-2  
Binary  
Search  
Looks  
OK.  
Closer to  
shipping.

What was  
achieved?  
\* tests  
that  
cover  
every  
path

\* default  
C-level  
checks  
such as  
array out  
of  
bounds,  
null de  
reference

i.e.  
FEWER  
BUGS

```
#include <klee/klee.h>
#include <stdio.h>
#include <assert.h>
#include <stdlib.h>

void print_data(int arr[], int size, int target) {
    printf("searching for %d in:\n", target);
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
}

klee@f88364af576a:~/klee_src/examples/binsrch$ ktest-tool klee-last/test00009.ktest
ktest file : 'klee-last/test00009.ktest'
args      : ['binsrch.bc']
num objects: 2
object 0: name: 'a'
object 0: size: 16
object 0: data: b'\x00\x00\x00\x06\x00\x00\x00\xfe\x00\x00\x00\x00\x00\x00\x03'
object 0: hex : 0x00000003600000fe00000000000003
object 0: text: ....6.....
object 1: name: 'x'
object 1: size: 4
object 1: data: b'\x00\x01\x00\x00'
object 1: hex : 0x00010000
object 1: int : 256
object 1: uint: 256
object 1: text: ....
klee@f88364af576a:~/klee_src/examples/binsrch$ gcc -DKLEEON -DPRINTON -I ../../include -L /home/klee/klee_build/lib binsrch.c -lkleeRuntst
klee@f88364af576a:~/klee_src/examples/binsrch$ KTEST_FILE=klee-last/test00001.ktest ./a.out
searching for 0 in:
[0, 0, 0, 0]
result = 1
klee@f88364af576a:~/klee_src/examples/binsrch$ KTEST_FILE=klee-last/test00002.ktest ./a.out
searching for 0 in:
[0, 16777216, 16777216, 16777216]
result = 0
klee@f88364af576a:~/klee_src/examples/binsrch$ KTEST_FILE=klee-last/test00009.ktest ./a.out
searching for 256 in:
[0, 54, 254, 50331648]
result = -1
klee@f88364af576a:~/klee_src/examples/binsrch$
```

# What does KLEE do? (Cadar's slides)

## KLEE

[OSDI 2008, Best Paper Award]

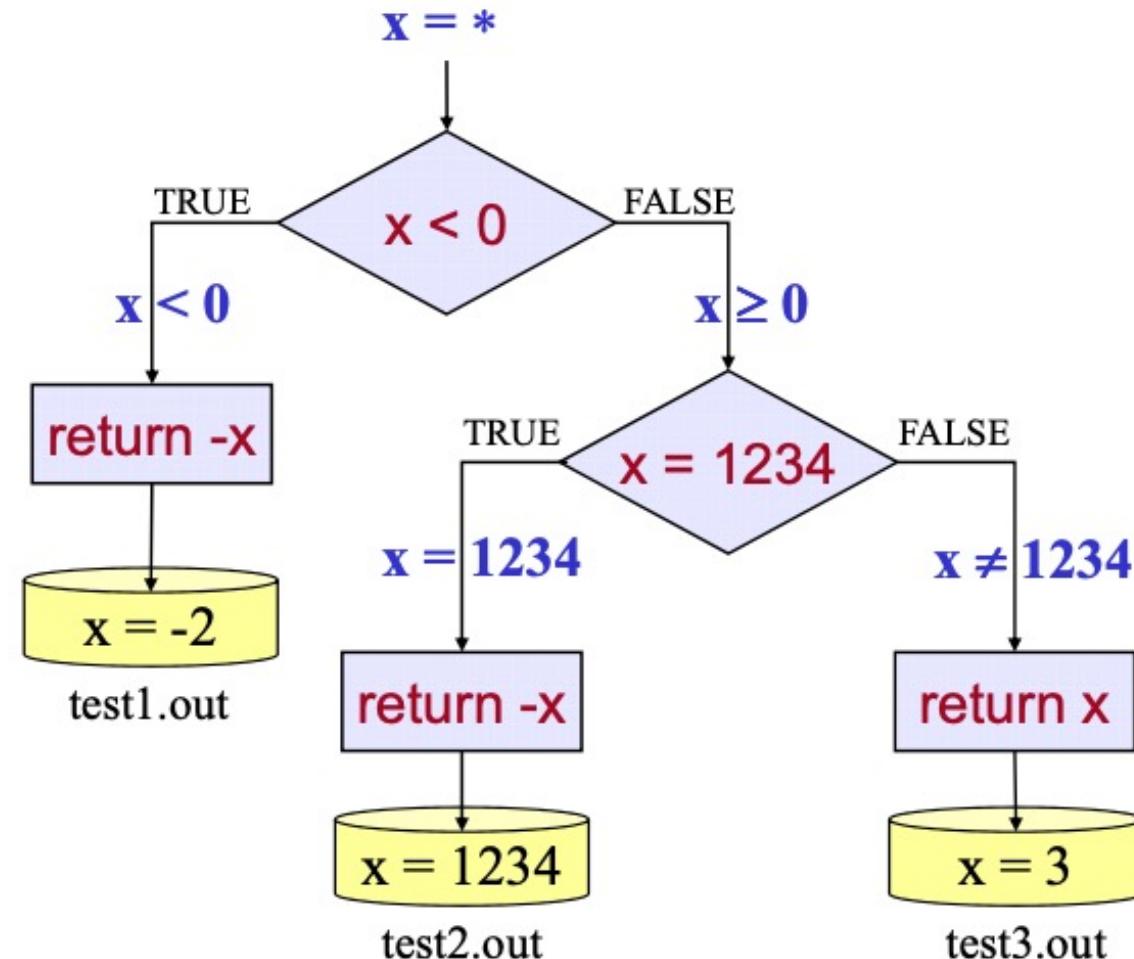
---

- Based on symbolic execution and constraint solving techniques
- Automatically generates high coverage test suites
  - Over 90% on average on ~160 user-level apps
- Finds deep bugs in complex systems programs
  - Including higher-level correctness ones

# How does KLEE work? (Cadar's slides)

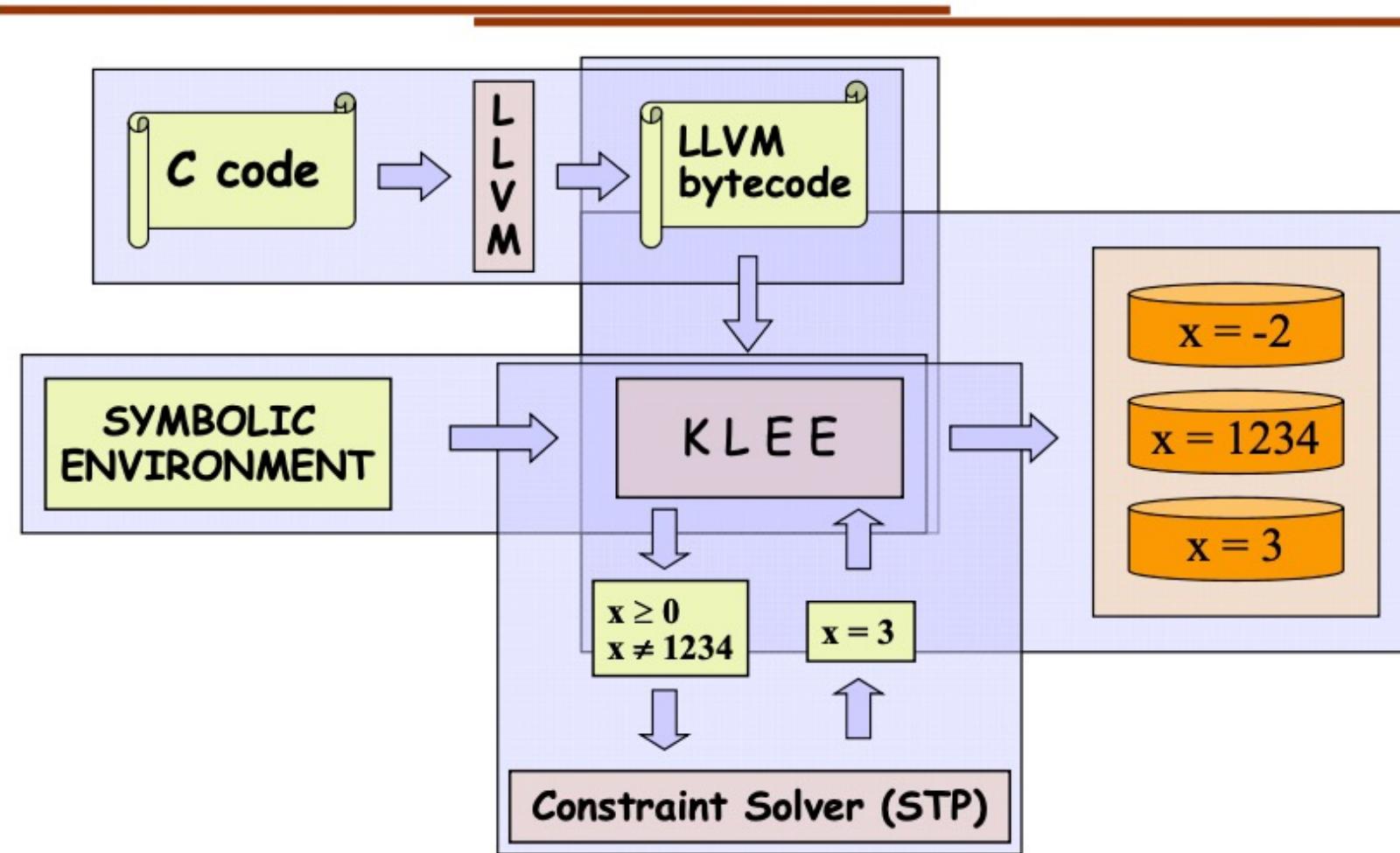
## Toy Example

```
int bad_abs(int x)
{
    if (x < 0)
        return -x;
    if (x == 1234)
        return -x;
    return x;
}
```



# How does KLEE work? (Cadar's slides)

## KLEE Architecture



# Engineering KLEE to be efficient (Cadar)

## Three Big Challenges

---

- Motivation
- Example and Basic Architecture
- ➡ • **Scalability Challenges**
  - Exponential number of paths
  - Expensive constraint solving
  - Interaction with environment
- Experimental Evaluation

# ConcFuzzer: a sanitizer guided hybrid fuzzing framework leveraging greybox fuzzing and concolic execution

Peng Li, Rundong Zhou, Yaohui Chen,

Yulong Zhang, Tao (Lenx) Wei

lipeng28@baidu.com



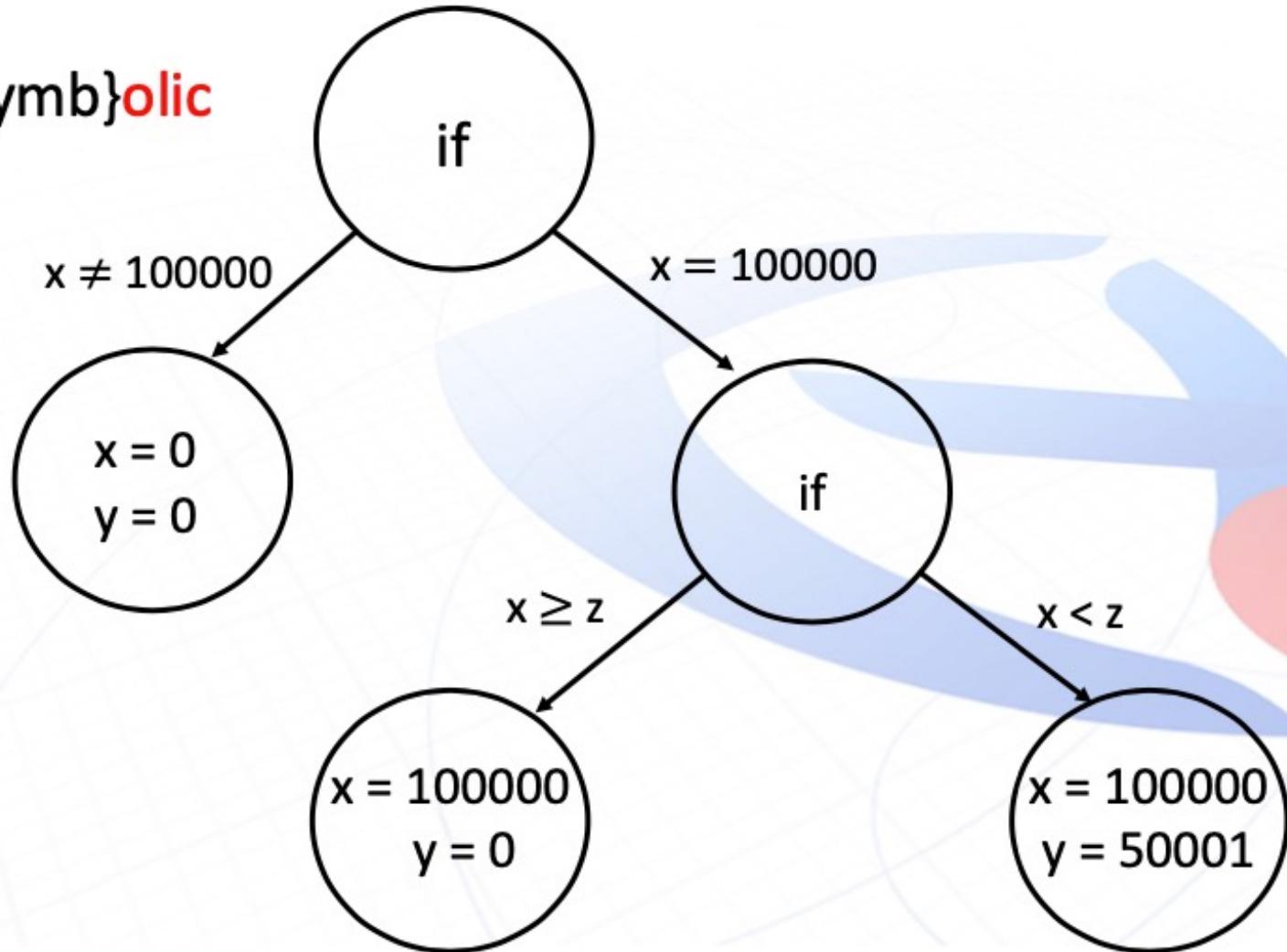
# Concolic Execution

How  
good is  
KLEE  
(Dr.  
Peng  
Li's  
slides)

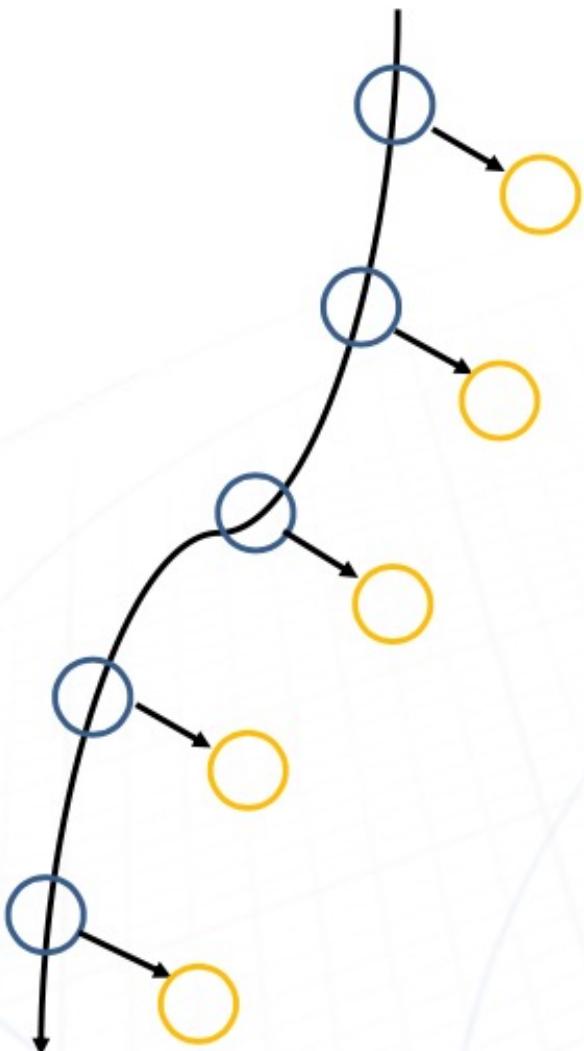
- Concolic = **Conc**rete + Symb**olic**

```
void foo (int x, int y) {  
    int z = 2*y;  
    if (x == 100000) {  
        if (x < z) {  
            /* error */  
            assert(0);  
        }  
    }  
}
```

Snippet of C code from wikipedia



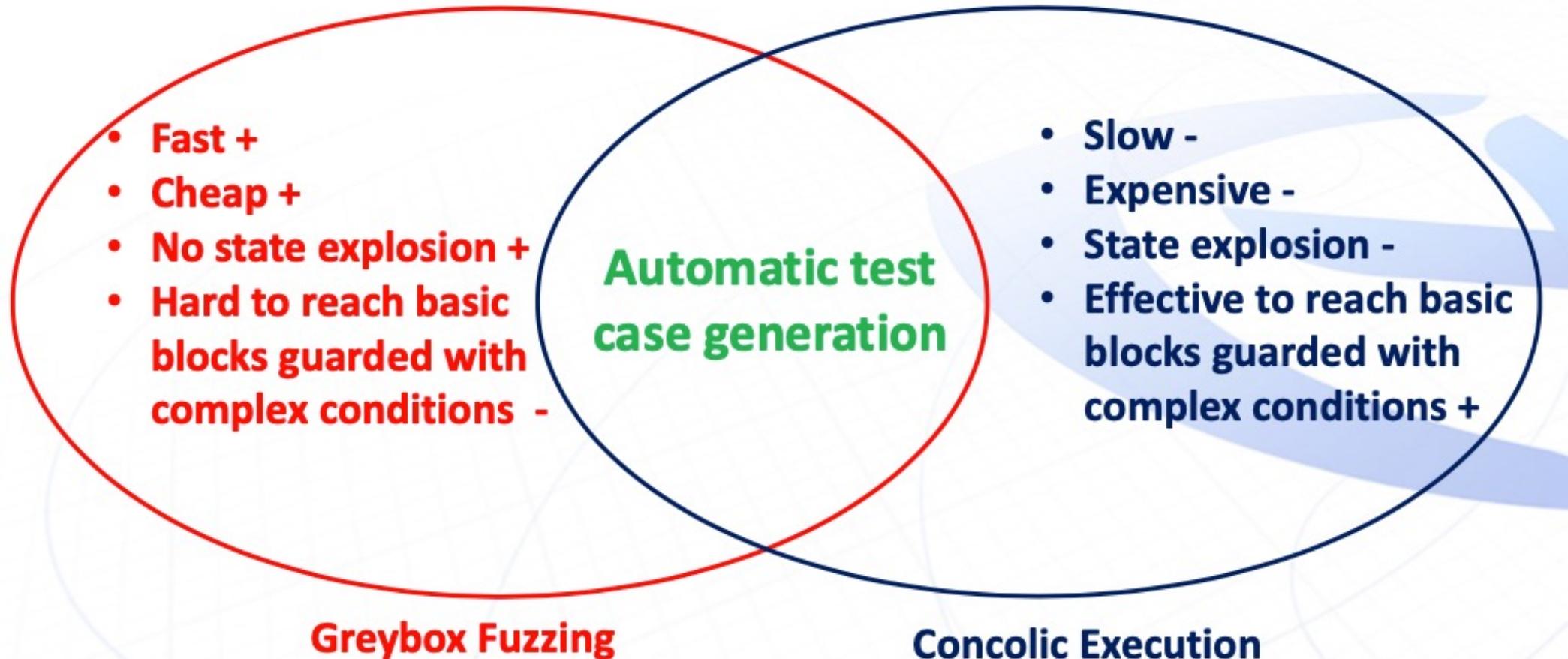
# Concolic Execution on top of KLEE



1. Delay constraint solving until the state get scheduled
2. If the state's path constraint is satisfiable, compute the input
3. Otherwise, discard the state

# Concolic execution

How  
good is  
KLEE  
(Dr.  
Peng  
Li's  
slides)



# MbedTLS X509 Certificate Parser

How  
good is  
KLEE  
(Dr.  
Peng  
Li's  
slides)

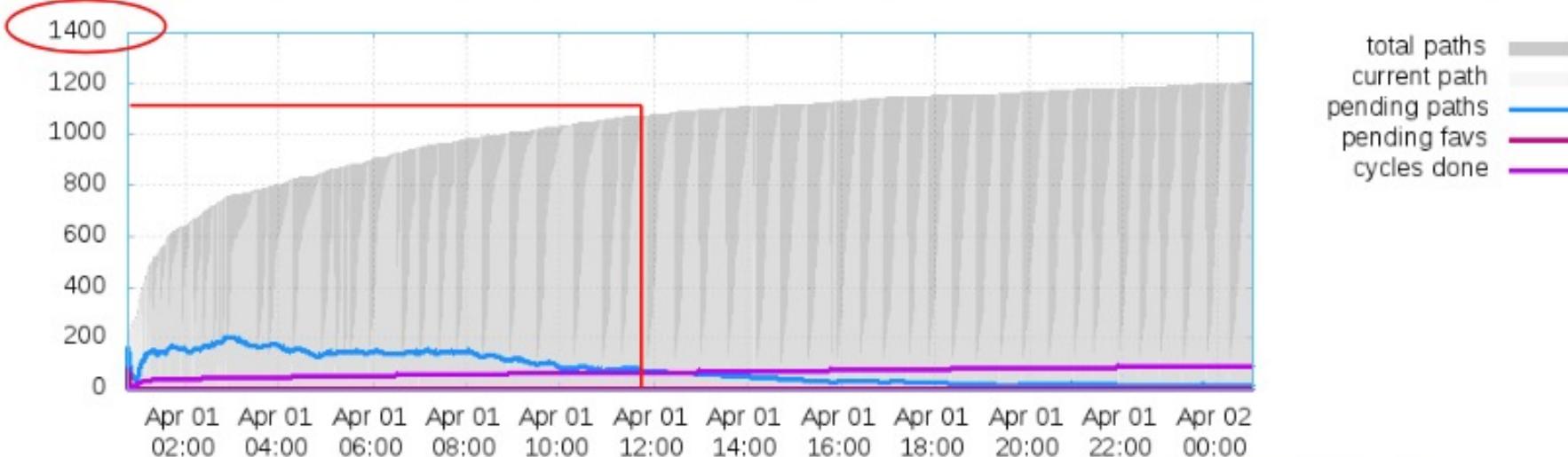


Fig 5. concfuzzer's 24 hours running results for mbedtls x509 certificate parser

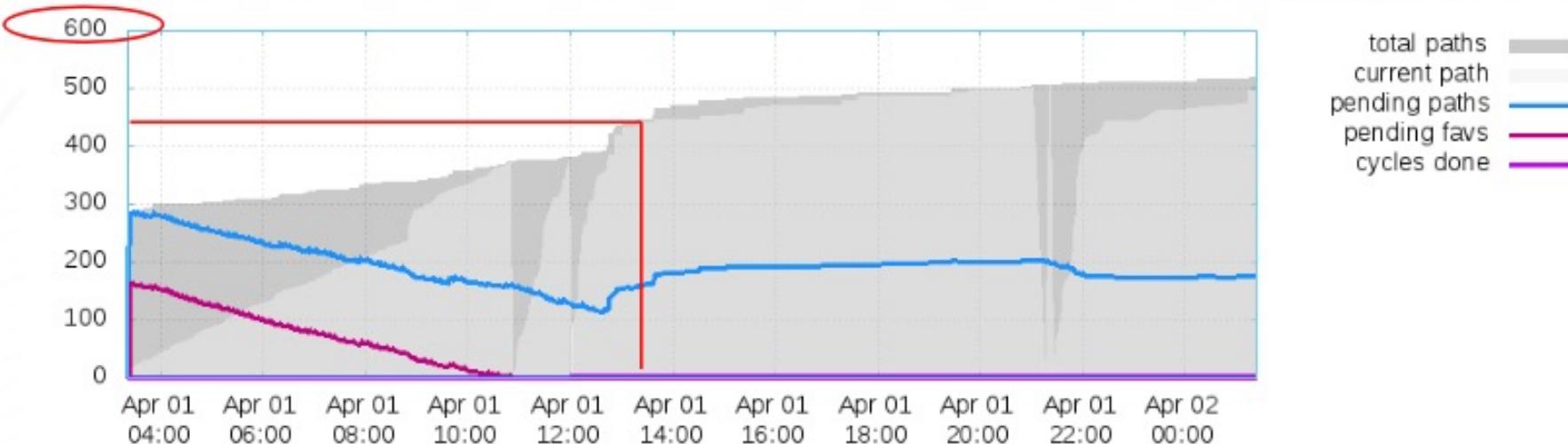


Fig 6. AFL's 24 hours running results for mbedtls x509 certificate parser

# djpeg (Libjpeg-9b)

How  
good is  
KLEE  
(Dr.  
Peng  
Li's  
slides)

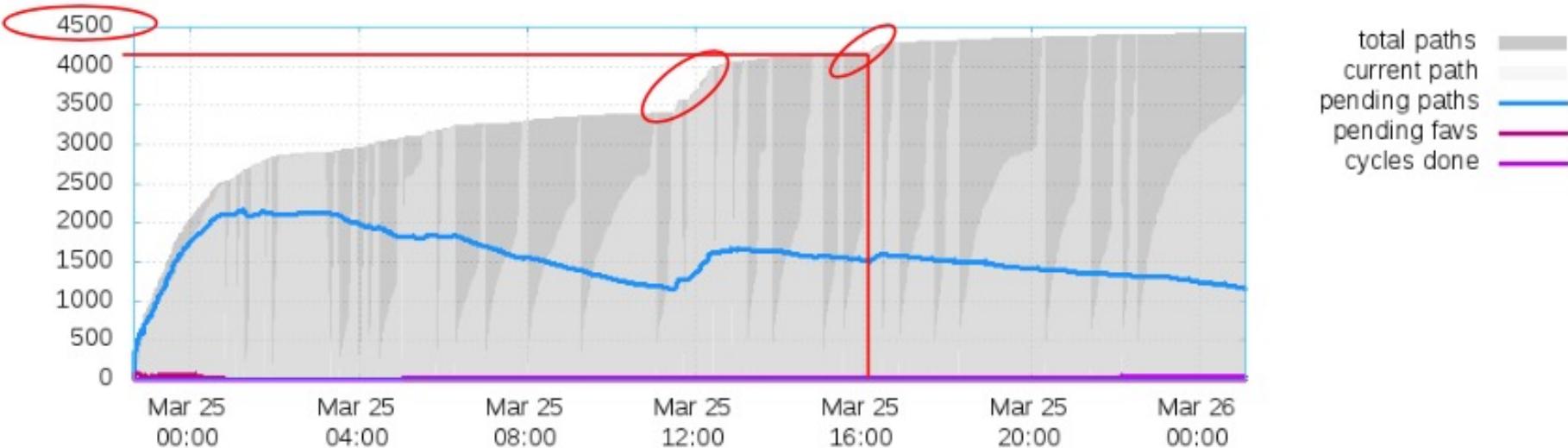


Fig 1. ConcFuzzer's 24 hours running results for djpeg

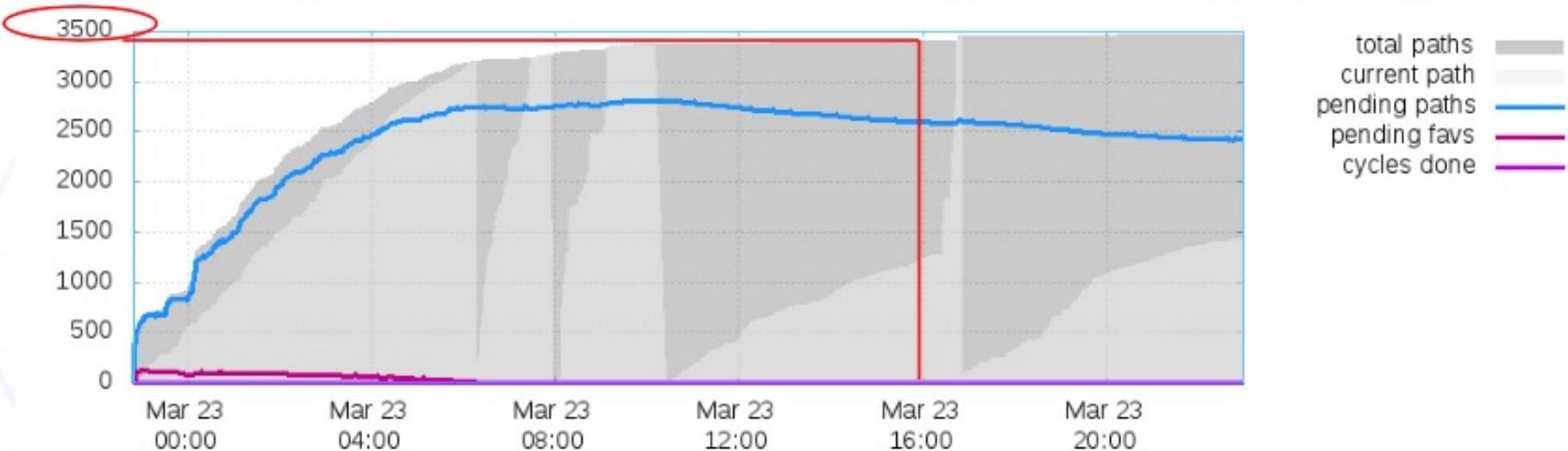


Fig 2. AFL's 24 hours running results for djpeg

Dr. Guodong Li's and Dr. Peng Li's FANTASTIC work on GKLEE follows

See also Guodong's work on PUG on the FSE 2010 publication site!

It was followed by GPUVerify, another fantastic verifier from Alastair Donaldson's groups at Imperial College, London

## Summary

This is a huge field now - symbolic execution

You get to study this in projects

Even the MSR python Dyn Symb Executor - good project!

# Correctness is central to Energy-Efficient Computing

The more performance per watt we can obtain (e.g. TeraFlop in 30W, Exaflop in 25 MW) ... **that is \$25 Million In electricity costs alone !!**

... the less we need to depend on dirty fuels,  
... the faster we can innovate in science and engineering



Intrepid supercomputer (Image courtesy of Argonne)



pogoprinciple.wordpress.com

# Emerging Heterogeneous Parallel Systems, and Role of Formal Methods in them



Problem Solving  
Environment based  
User Applications

Problem-Solving  
Environments  
e.g. Uintah, Charm++,  
ADLB

(1) Formal  
Methods at  
the User  
Application  
Level

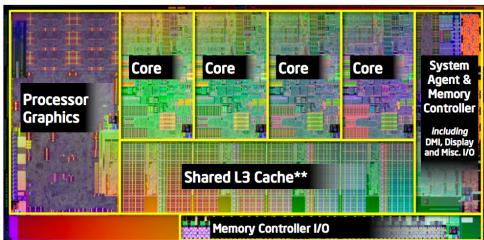


Infiniband style interconnect

High Performance  
MPI Libraries

Concurrent  
Data Structures

(2) Formal  
Dynamic  
Methods for  
MPI



Sandybridge (courtesy anandtech.com)



AMD Fusion APU

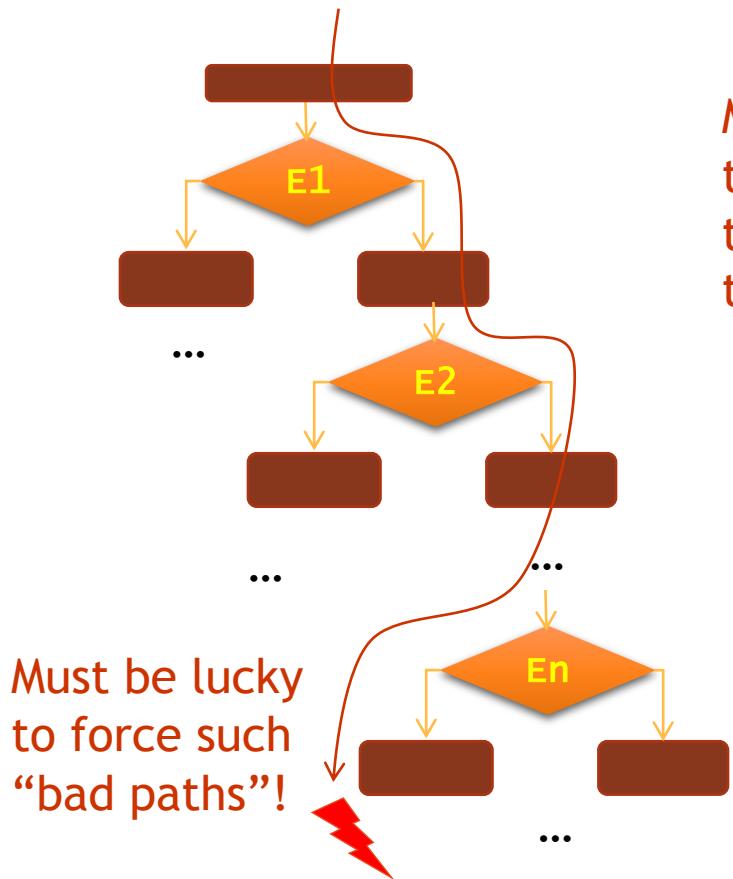


Geoforce GTX 480 (Nvidia)

(3) Formal  
Analysis for  
CUDA  
Programs

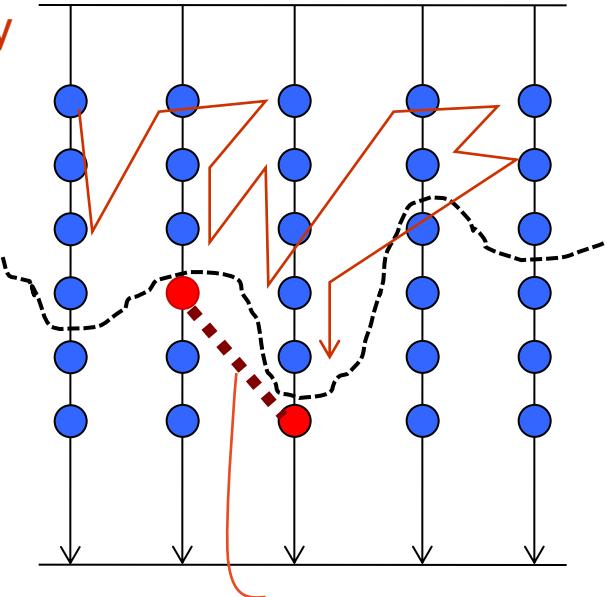
# Why conventional testing is inadequate

Exponential number of paths;  
**the bug** may be deeply hidden



Must be lucky  
to schedule  
threads in  
this manner!

Exponential number of  
schedules; **the bug** may be  
triggered only by  
one particular interleaving



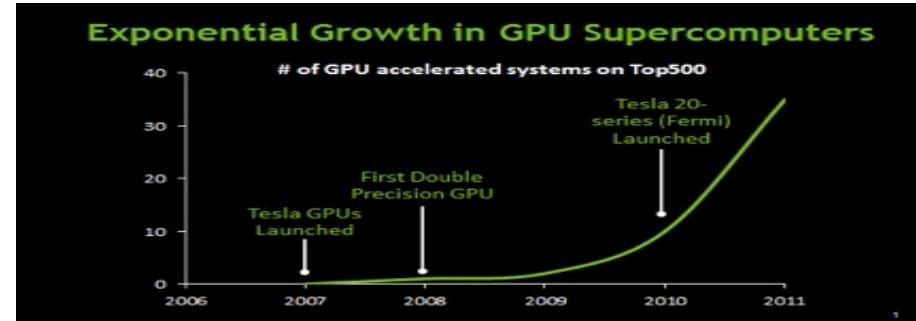
Some bad interaction, such as  
a data race may depend on  
the schedule...

# GPU-based Computing

- About 40 of the top 500 machines were (in 2012) GPU-based



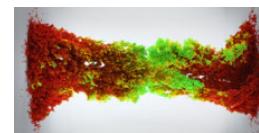
(courtesy of Nvidia,  
[www.engadget.com](http://www.engadget.com))



- Personal supercomputers used for scientific research (biology, physics, ...) increasingly based on GPUs



(courtesy of AMD)



(courtesy of Nvidia)

# GKLEE (see PPoPP 2012)

```
__input__ int *values = (int *)malloc(sizeof(int) * NUM);

klee_make_symbolic(values, sizeof(int)*NUM, "values");

int *dvalues;
cudaMalloc((void **)&dvalues, sizeof(int) * NUM);
cudaMemcpy(dvalues, values, sizeof(int) * NUM, cudaMemcpyHostToDevice);

BitonicKernel <<< ... >>> (dvalues);
```

C++ CUDA Programs  
with Symbolic Variable  
Declarations

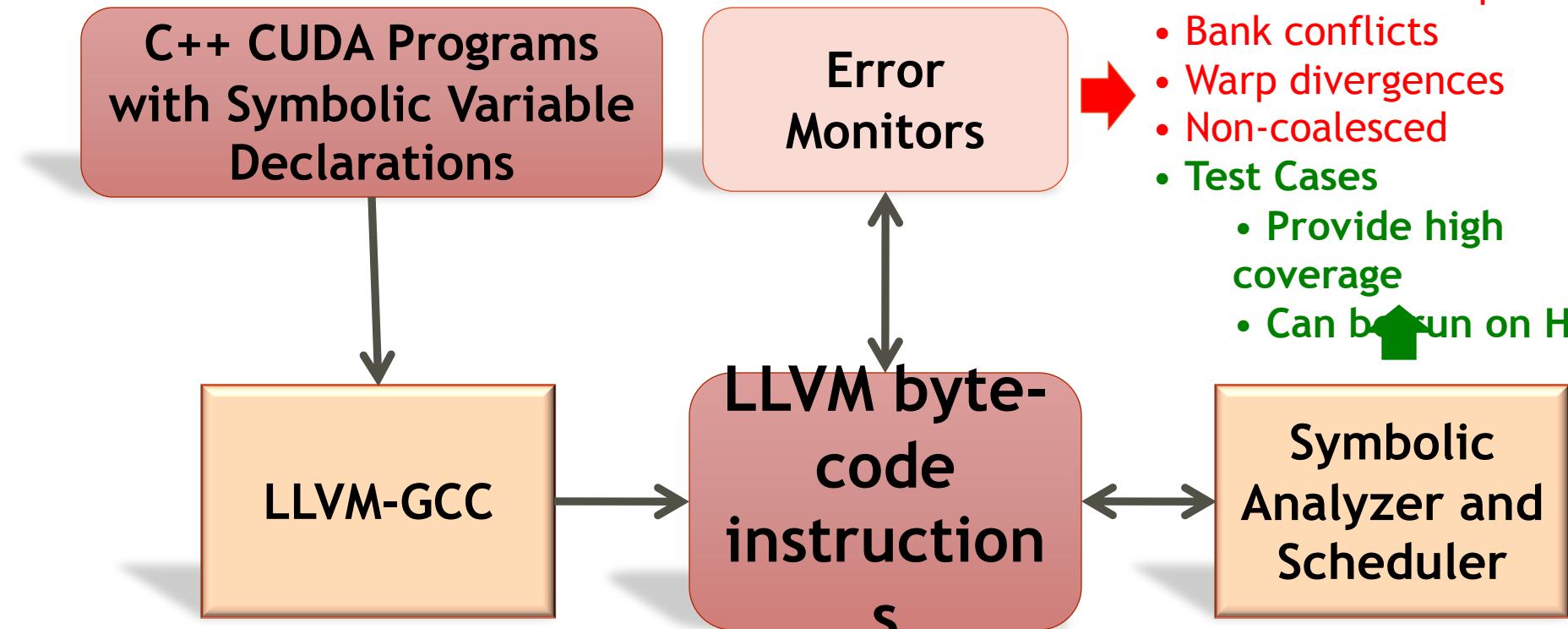
Error  
Monitors

- Deadlocks
- Data races
- Concrete test inputs
- Bank conflicts
- Warp divergences
- Non-coalesced
- Test Cases
  - Provide high coverage
  - Can be run on HW

LLVM-GCC

LLVM byte-  
code  
instruction s

Symbolic  
Analyzer and  
Scheduler



# Our Formal Verification of CUDA builds on top of Sequential Program Verification by KLEE [OSDI '08]

```
#include "stdio.h"
#include "cutil.h"
#include "string.h"
#include "klee.h"

#define STRLEN 5
int main(){
    char item;
    char str[STRLEN];
    int lo=0;
    int hi;

    int mid, found=0;
    // printf("Please give char to be searched: \n");
    // scanf("%c", &item);
    klee_make_symbolic(&item, sizeof(item), "item");
    // printf("Please give string within which to search: \n");
    // scanf("%s", str);
    klee_make_symbolic(str, sizeof(str), "str");
    klee_assume(str[0] <= str[1]);
    klee_assume(str[1] <= str[2]);
    klee_assume(str[2] <= str[3]);
    klee_assume(str[3] <= str[4]);
    str[4] = '\0';
    hi= 4; //strlen(str)-1; // strlen ignores null at the end of string
           // hi points to last char in a 0 based array

    while(!found && lo <= hi) {
        mid = (lo+hi)/2;

        if (item == str[mid])
            { printf("*");
             found = 1; }
        else
            if (item < str[mid])
                { hi = mid-1;
                  printf("L"); }
            else
                { lo = mid+1;
                  printf("H"); }

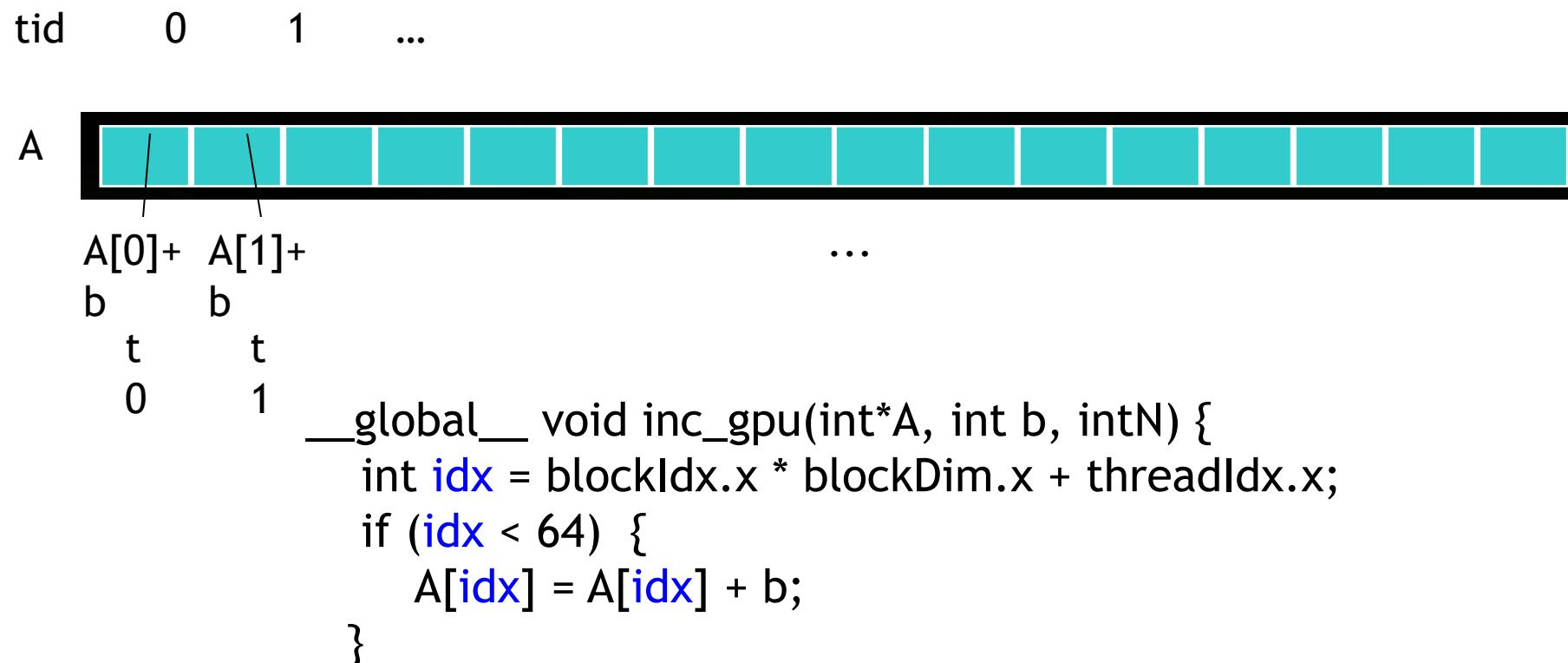
        if (found) printf("found\n"); //printf("\nitem %c found at posn %d\n", item, mid);
        else printf("not found\n"); //printf("\nitem %c not in str %s\n", item, str);
    }
    return found;
}
```

A simple Binary Search being prepared for formal symbolic analysis  
This will force path coverage !!

# Concurrency errors

# Example: Increment Array Elements

## Increment N-element array A by scalar b



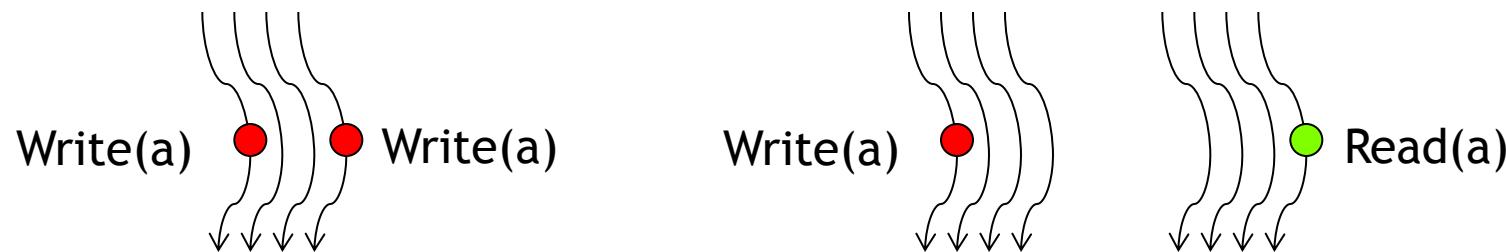
# Data Races in GPU Programs

The usual definition of a race:

“Two accesses, one of which is a write, that occur without a happens-before edge between them.”

In GPUs, the happens-before is provided by

- Barriers (`_syncthreads()`)
- Atomics



# Illustration of Race

Increment N-element vector A by scalar b

tid 0 1



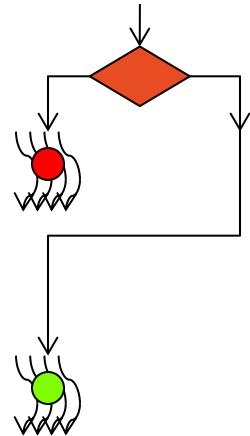
```
__global__ void inc_gpu(int*A, int b, int N) {  
    int idx = blockIdx.x * blockDim.x +  
    threadIdx.x;
```

```
        if (idx < 64) {  
            A[idx] = A[(idx - 1 + 64) % 64] + b;  
        }  
    }  
    RACE!  
    t63 t0
```

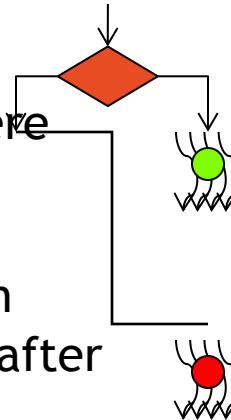
t0: read A[63]  
t63: write A[63]

# Porting Race: A Race-type Identified by us

- Some hardware platforms may evaluate “then” before “else” – others may reverse this order



If we have a divergent warp where a variable is **READ** in one path and **WRITTEN** in another path, then depending on the execution order, the **READ** may be before/after the **WRITE**



# Example of Porting Race

```
#include <stdio.h>

#define NUM 32

__shared__ int v[NUM];

__global__ void PortingRace() {
    if (threadIdx.x % 2) {
        v[threadIdx.x] = v[ (NUM + threadIdx.x - 1) % NUM ] + 1; // W:even;
R:odd
    }
    else {
        v[threadIdx.x] = v[ (NUM + threadIdx.x + 1) % NUM ] - 1; // W:odd;
R:even
    }
}
int main() {
    PortingRace<<<1,NUM>>>();
    return 0;
}
```

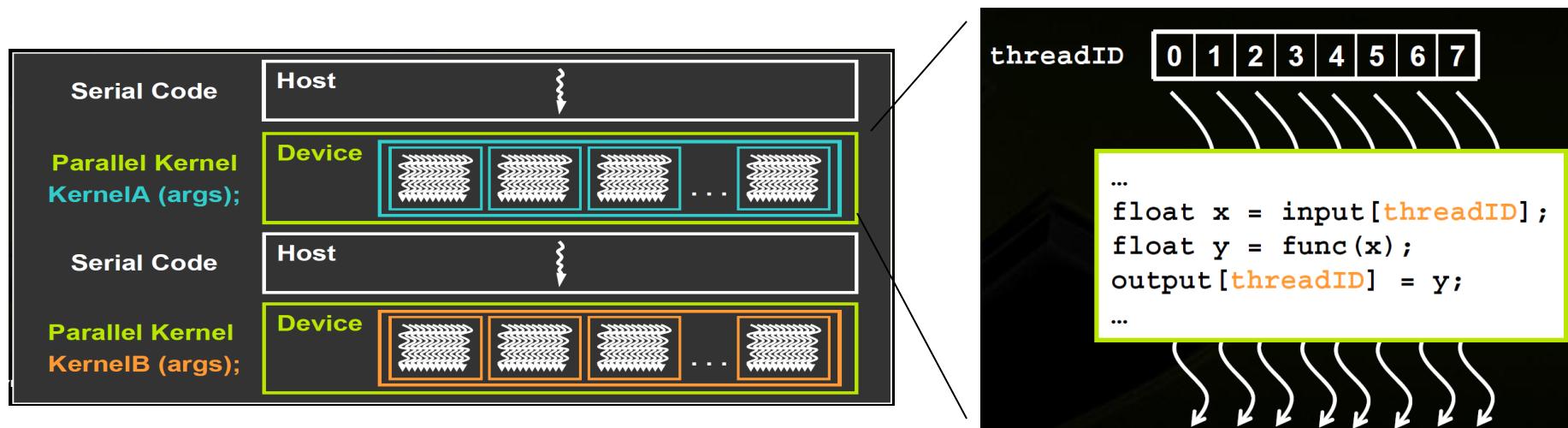
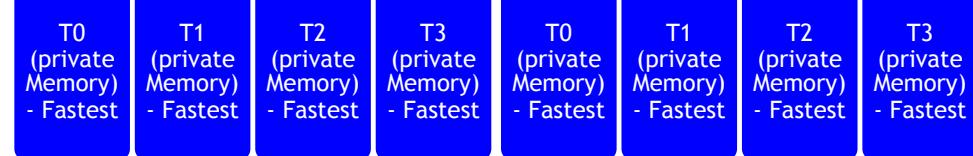
# CUDA-based GPU Programming Basics

- A simple dialect of C++ with CUDA directives
- Synchronization within Block through Barriers
- `(__syncthread())`
- Threads within block scheduled in *warps*
- **Don't rely on warps for correctness!**

Device Global Memory (shared between blocks) - Slowest

Block (Shared Memory) - Faster

Block (Shared Memory) - Faster



# Conventional GPU Debuggers are Inadequate

- Based on platform testing
- Full generality of executions not reflected
- Hard to cause races (input selection)
- Hard to observe races
  - vector-clock based solutions that run on GPUs exist
  - (last time we checked) only for shared memory races

# Some of the GPU Program Bugs

- Data races
  - Cause unpredictable outputs
  - Cause compilers to misbehave
- Incorrectly used synchronizations
  - Syncthreads not used with textual barriers
  - Syncthreads causing deadlocks
- Misunderstood memory consistency models
  - When updates are visible across threads
- Incorrectly used atomics (synchronization)
  - Due to incorrect synchronization, invariants get broken
- Erroneous computational results - esp. with floating-point
  - Due to unpredictable computational order, results may diverge

# Why are Data Races Highly Problematic?

- Testing is seldom conclusive
  - Must infer a race indirectly (e.g. corrupted results)
- Races manifest when code is ported or optimized
  - Scheduling can change
- Data races make compiled results suspect
  - Compilers optimize assuming the absence of races in the most general setting

# Synchronization Primitives in CUDA

Device Global Memory (shared between blocks) - Slowest

Block (Shared Memory) - Faster

Block (Shared Memory) - Faster

T0  
(private Memory)  
- Fastest

T1  
(private Memory)  
- Fastest

T2  
(private Memory)  
- Fastest

T3  
(private Memory)  
- Fastest

T0  
(private Memory)  
- Fastest

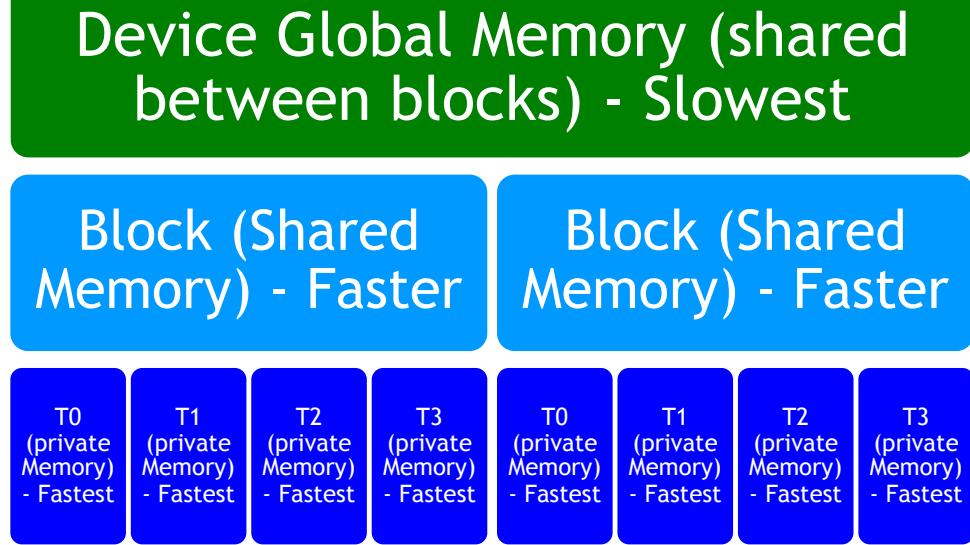
T1  
(private Memory)  
- Fastest

T2  
(private Memory)  
- Fastest

T3  
(private Memory)  
- Fastest

```
__kernel__ void deadlock() {
    if (threadIdx.x > 0) {
        v[threadIdx.x]++;
    }
    else {
        v[threadIdx.x]--;
    }
    __syncthreads(); // Barrier
}
```

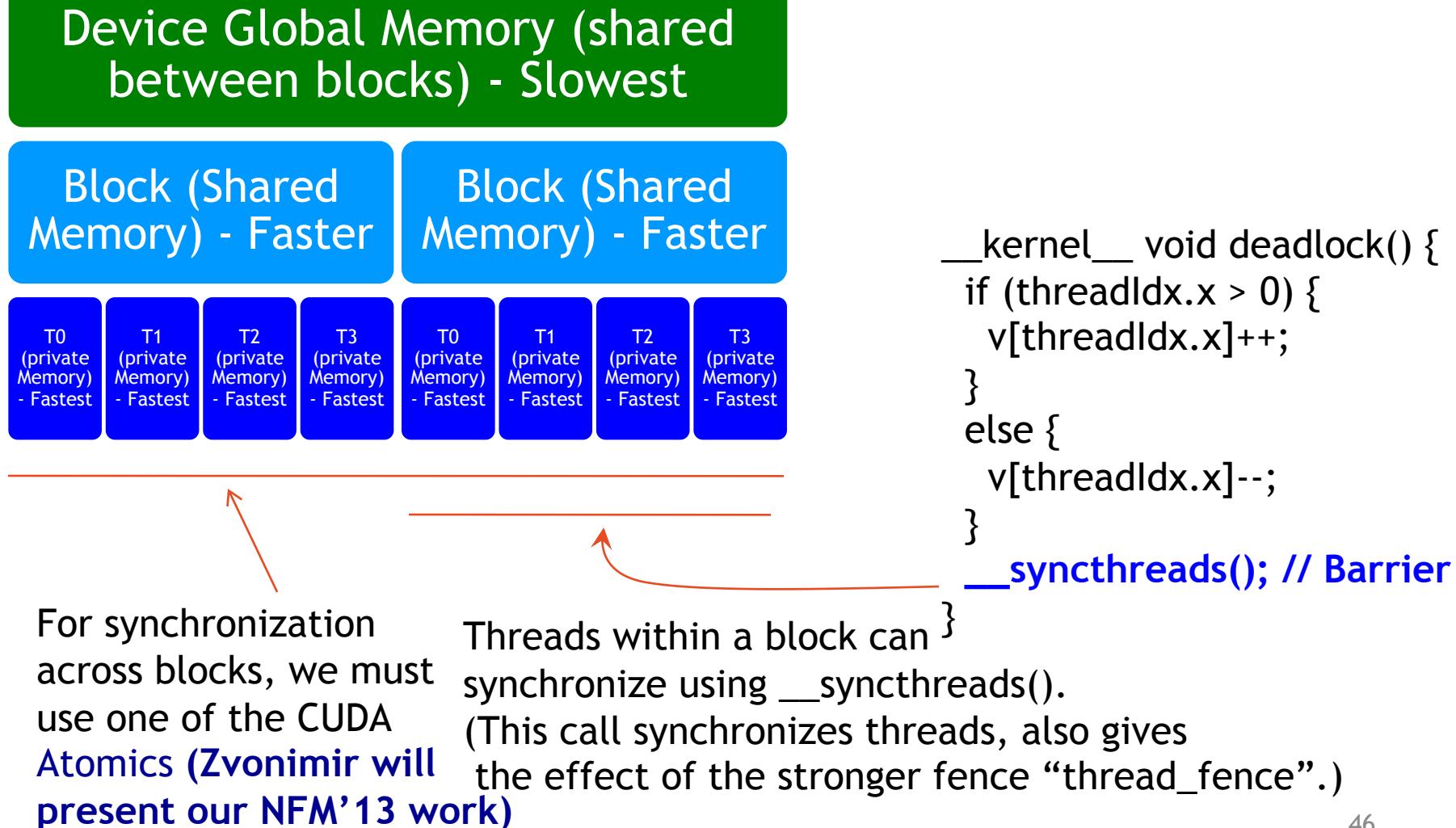
# Synchronization Primitives in CUDA



Threads within a block can synchronize using `__syncthreads()`.  
(This call synchronizes threads, also gives the effect of the stronger fence “`thread_fence`”.)

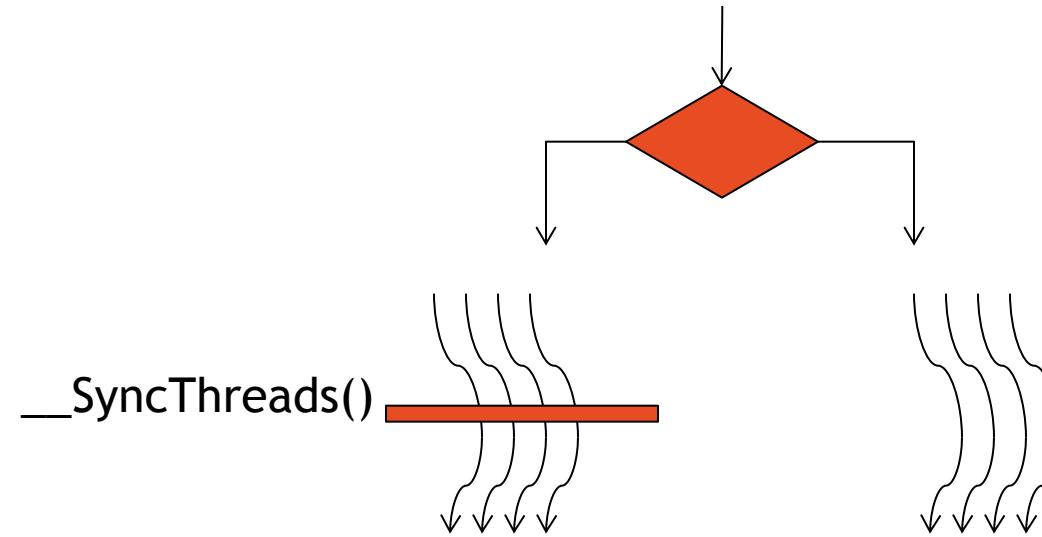
```
__kernel__ void deadlock() {
    if (threadIdx.x > 0) {
        v[threadIdx.x]++;
    }
    else {
        v[threadIdx.x]--;
    }
    __syncthreads(); // Barrier
}
```

# Synchronization Primitives in CUDA



# Deadlocks caused by misused `_syncthreads()`

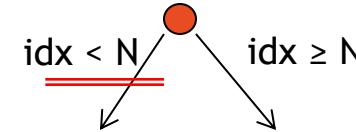
- Cause a “hang” or unspecified behavior



# Illustration of “Deadlock” (conceptual deadlock; behavior is really undefined)

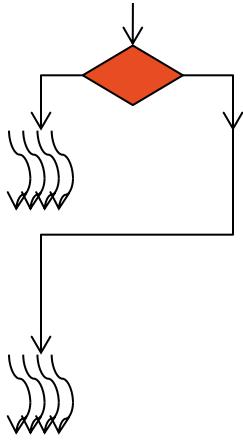
```
__global__ void kernelName (args) {  
  
    int idx = blockIdx.x * blockDim.x +  
    threadIdx.x;  
  
    if (idx == 3) {  
        ....  
        __syncthreads();  
    }  
}
```

Suffers from Textually  
Non-aligned Barriers  
(example Deadlock1.C)



```
__kernel__ void deadlock() {  
    if (threadIdx.x > 0) {  
        v[threadIdx.x]++;  
        __syncthreads();  
    }  
    else {  
        v[threadIdx.x]--;  
        __syncthreads();  
    }  
}
```

# Warp Divergence



```
__global__ void kernelName (args) {
```

```
    int idx = blockIdx.x * blockDim.x +  
    threadIdx.x;
```

```
    if ( odd(idx) ) {
```

<Statements1>

```
} else {
```

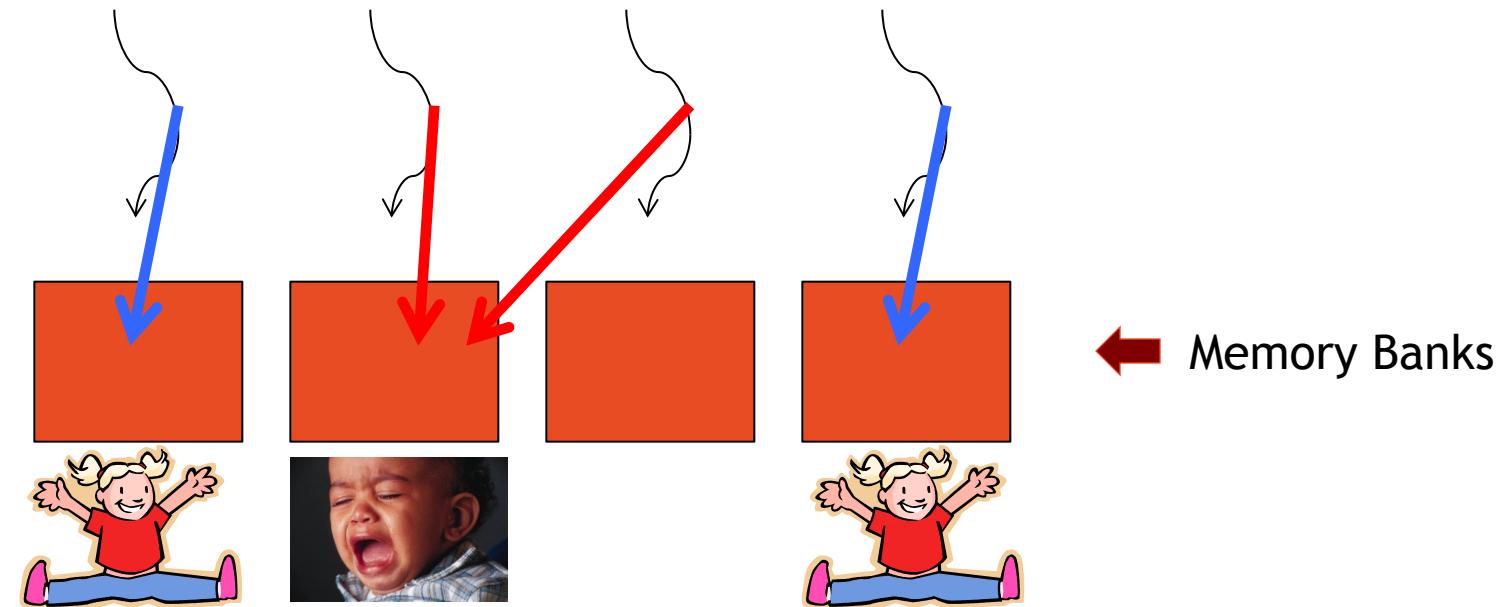
<Statements2>

}

This is a performance bug - but can be detected through formal symbolic analysis

# Memory-bank Conflicts

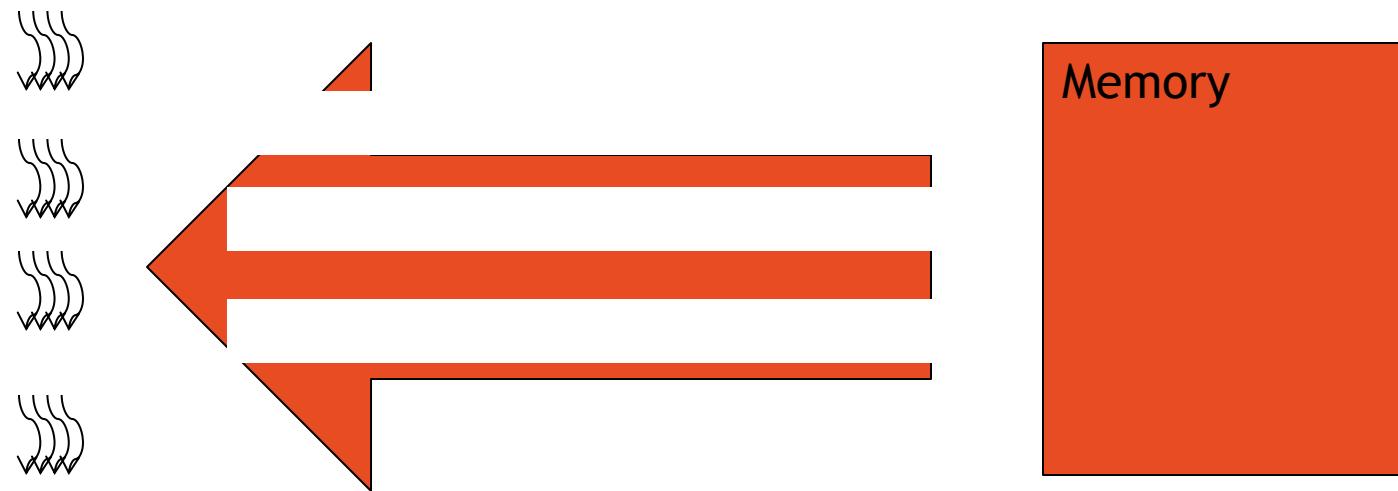
Happen when two threads generate addresses that fall into the same memory bank



This is a performance bug - but can be detected through formal symbolic analysis

# Non-coalesced Memory Fetches

Happen when threads generate global memory addresses that do not fully utilize the bus



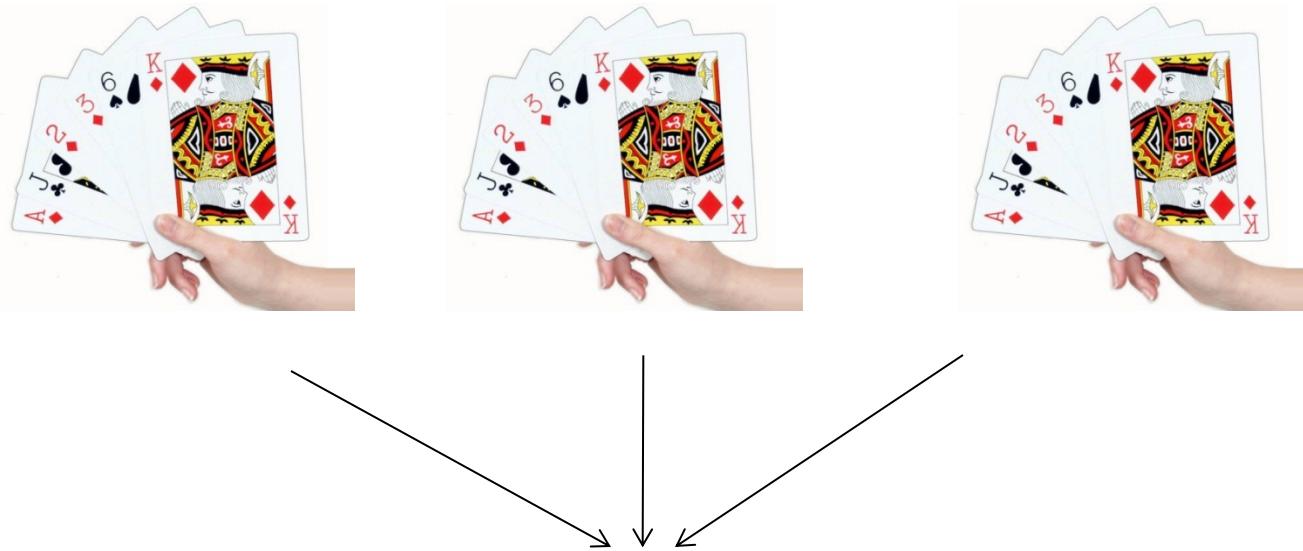
This is a performance bug - but can be detected through formal symbolic analysis  
(10 x more severe than bank conflicts)

# Two approaches to modeling

- Through interleaving
  - Followed in most works
- Through lock-step synchronous execution
  - Pioneered in GPUVerify

# How bad is the interleaving problem?

It is like shuffling decks of cards



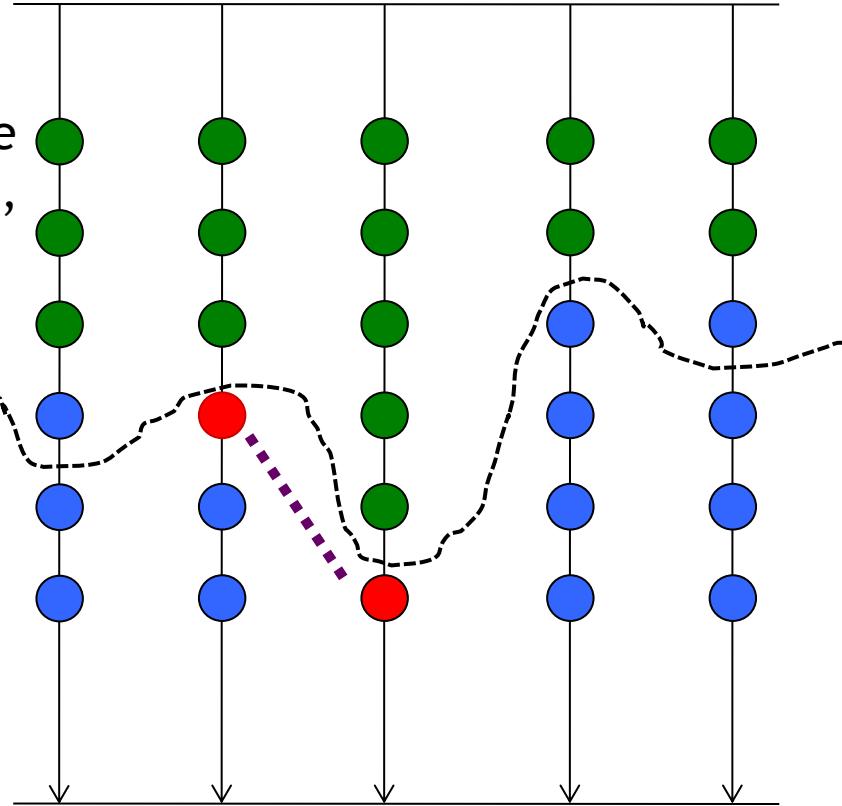
>  $6 * 10^{14}$  interleavings for 5 threads with 5 instructions

# Avoiding Interleaving Explosion

# Solution to the interleaving problem: Find *representative* interleavings

For Example:

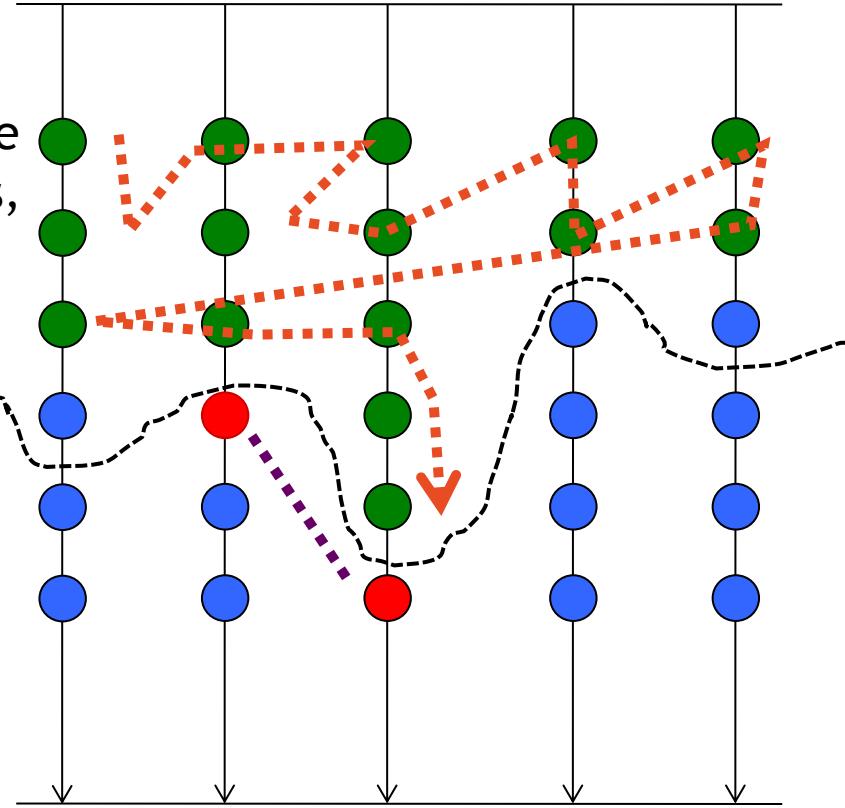
If the green dots are local thread actions,  
then  
all schedules  
that arrive  
at the “cut line”  
are equivalent!



# Solution to the interleaving problem: Find *representative* interleavings

For Example:

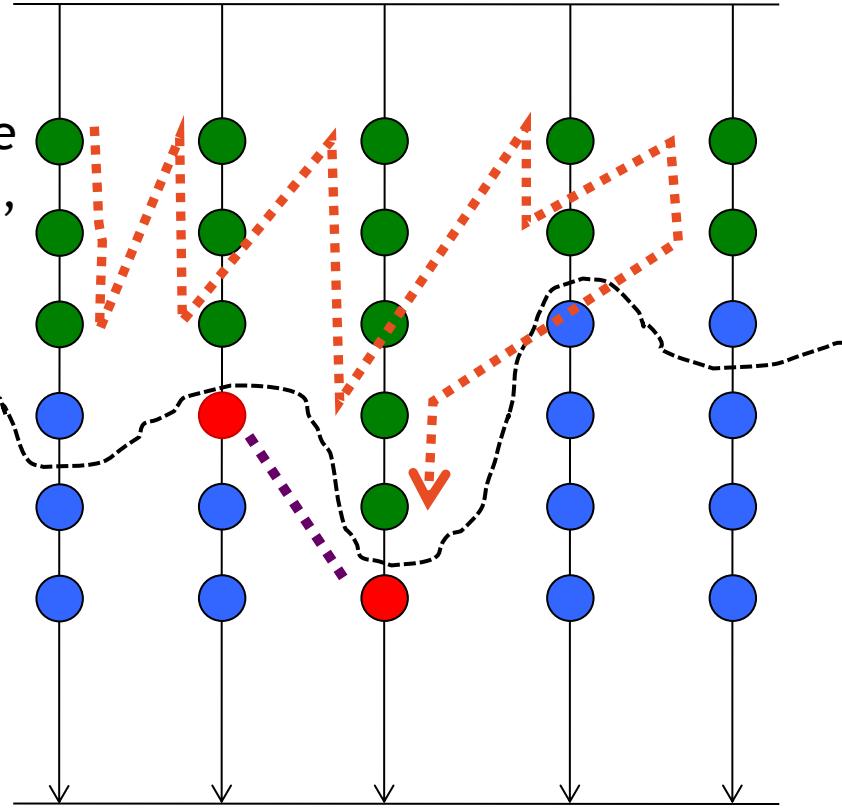
If the green dots are local thread actions,  
then  
all schedules  
that arrive  
at the “cut line”  
are equivalent!



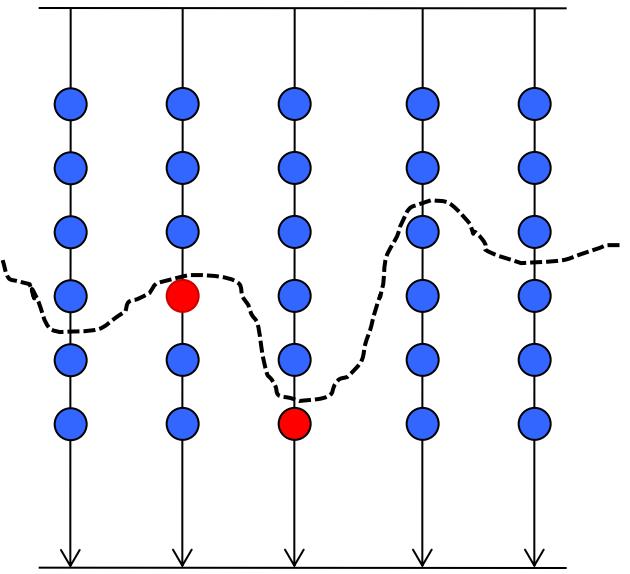
# Solution to the interleaving problem: Find *representative* interleavings

For Example:

If the green dots are local thread actions,  
then  
all schedules  
that arrive  
at the “cut line”  
are equivalent!

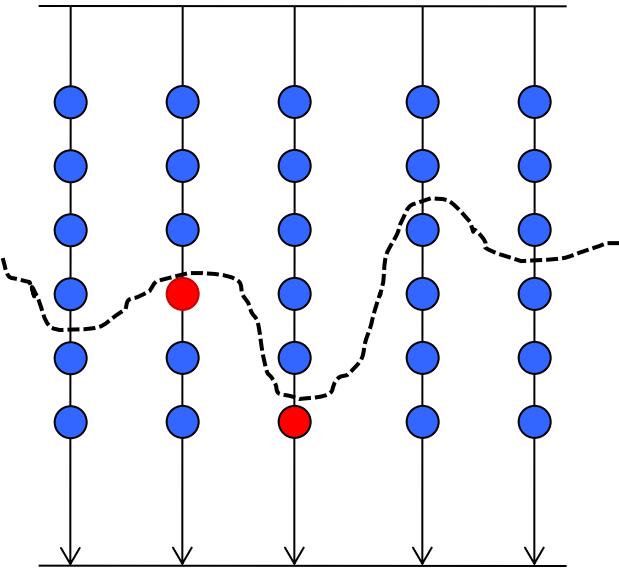


# GKLEE Avoids Examining Exp. Schedules !!

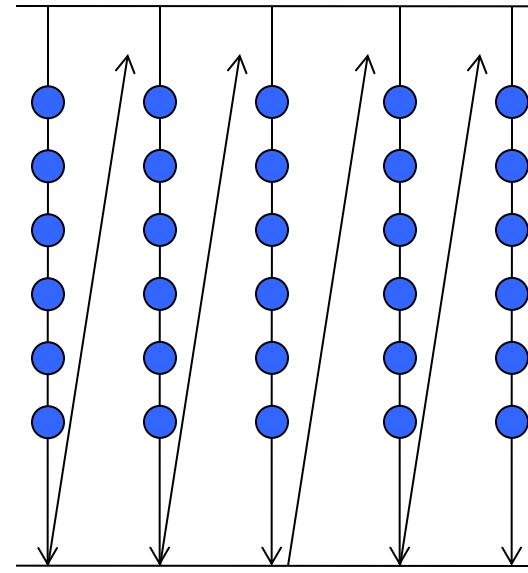


Instead of considering all  
Schedules and  
All Potential Races...

# GKLEE Avoids Examining Exp. Schedules !!



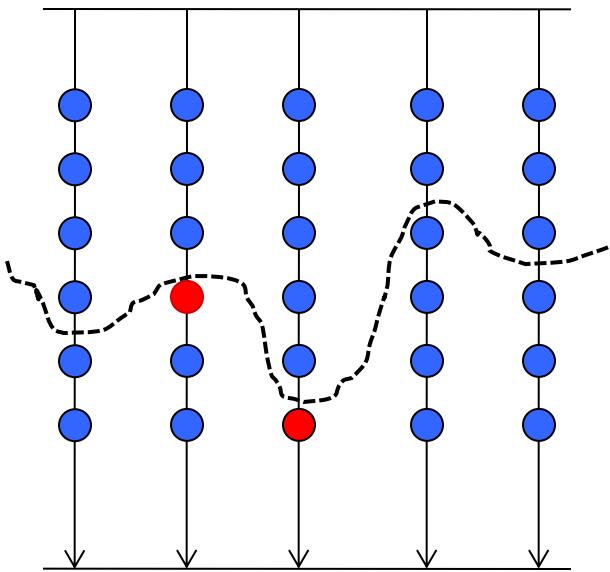
Instead of considering all  
Schedules and  
All Potential Races...



Consider JUST THIS SINGLE  
CANONICAL SCHEDULE !!

Folk Theorem (proved in our paper):  
“We will find **A RACE**  
If there is ANY race” !!

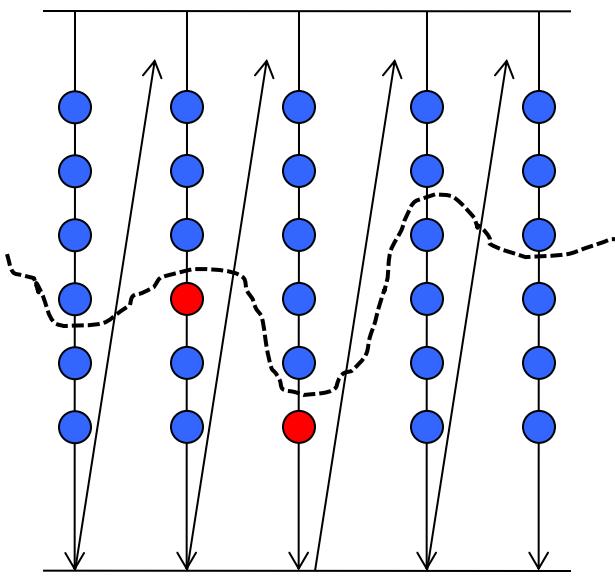
# Why does a sequential run suffice?



If the red pair is the only race, then it occurs after the cut-line

So all the ways to reach the cut-line are EQUIVALENT !!

# So this sequential run is equivalent too.



You my complain saying:  
“But, the sequential run  
Inches Past the Cutline !!”

But hey, you told me that  
this is the ONLY race!

If there were some other  
race encountered before,  
then that will be caught.

... there is a “first race”  
and that will be caught !

# Concolic execution helps find witnesses to data races

```
__global__ void histogram64Kernel(unsigned *d_Result, unsigned *d_Data, int  
dataN) {  
    const int threadPos = ((threadIdx.x & (~63)) >> 0)  
                      | ((threadIdx.x & 15) << 2)  
                      | ((threadIdx.x & 48) >> 4);  
    ...  
    __syncthreads();  
    for (int pos = IMUL(blockIdx.x, blockDim.x) + threadIdx.x; pos < dataN;  
         pos += IMUL(blockDim.x, gridDim.x)) {  
        unsigned data4 = d_Data[pos];  
        ...  
        addData64(s_Hist, threadPos, (data4 >> 26) & 0x3FU); }  
        __syncthreads(); ...  
    }  
    inline void addData64(unsigned char *s_Hist, int threadPos, unsigned int data)  
    { s_Hist[ threadPos + IMUL(data, THREAD_N) ]++; }
```

“GKLEE: Is there a Race ?”

# Concolic execution helps find witnesses to data races

```
__global__ void histogram64Kernel(unsigned *d_Result, unsigned *d_Data, int dataN) {
    const int threadPos = ((threadIdx.x & (~63)) >> 0)
        | ((threadIdx.x & 15) << 2)
        | ((threadIdx.x & 48) >> 4);

    ...
    __syncthreads();
    for (int pos = IMUL(blockIdx.x, blockDim.x) + threadIdx.x; pos < dataN;
         pos += IMUL(blockDim.x, gridDim.x)) {
        unsigned data4 = d_Data[pos];
        ...
        addData64(s_Hist, threadPos, (data4 >> 26) & 0x3FU); }
    __syncthreads(); ...
}

inline void addData64(unsigned char *s_Hist, int threadPos, unsigned int data)
{ s_Hist[ threadPos + IMUL(data, THREAD_N) ]++; }
```

Threads 5 and and 13 have a WW race

when d\_Data[5] = 0x04040404 and d\_Data[13] =

GKLEE

0.

## Summary

This is a huge field now - symbolic execution

You get to study this in projects

Even the MSR python Dyn Symb Executor - good project!