TERN:
Stable Deterministic Multithreading through Schedule Memoization

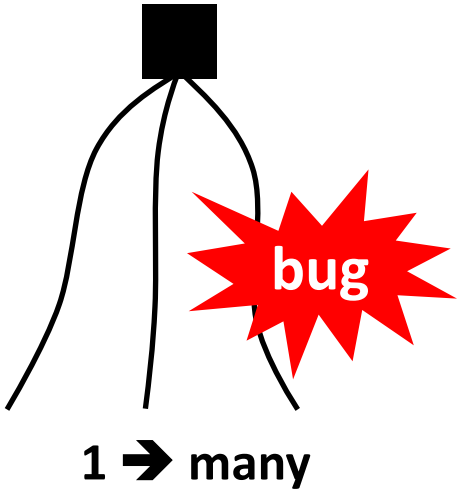
Heming Cui
Jingyue Wu
Chia-che Tsai
Junfeng Yang

Computer Science
Columbia University
New York, NY, USA

Nondeterministic Execution

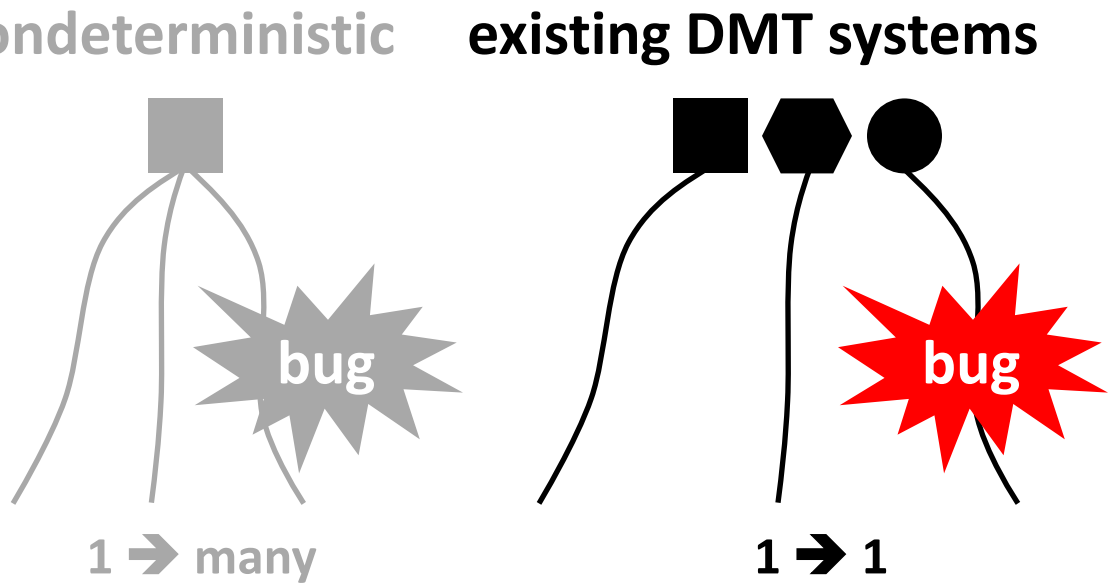
- Same input → many schedules
- Problem: different runs may show different behaviors, even on the same inputs

nondeterministic



Deterministic Multhreading (DMT)

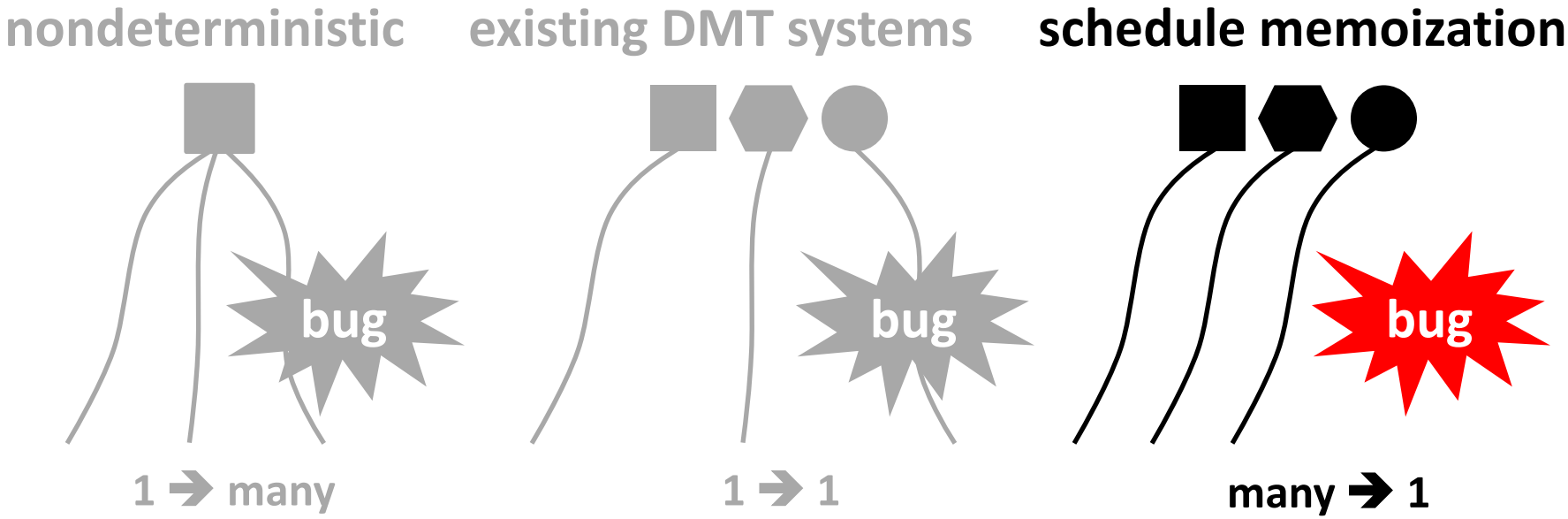
- Same input → same schedule
  - [DMP ASPLOS '09], [KENDO ASPLOS '09], [COREDET ASPLOS '10], [dOS OSDI '10]
- Problem: minor input change → very different schedule



Confirmed in experiments

Schedule Memoization

- Many inputs → one schedule
  - Memoize schedules and reuse them on future inputs
- ***Stability***: repeat familiar schedules
  - Big benefit: avoid possible bugs in unknown schedules



TERN: the First Stable DMT System

- Run on Linux as user-space schedulers
- To memoize a new schedule
  - Memoize total order of synch operations as schedule
    - Race-free ones for determinism [RecPlay TOCS]
  - Track input constraints required to reuse schedule
    - symbolic execution [KLEE OSDI '08]
- To reuse a schedule
  - Check input against memoized input constraints
  - If satisfies, enforce same synchronization order

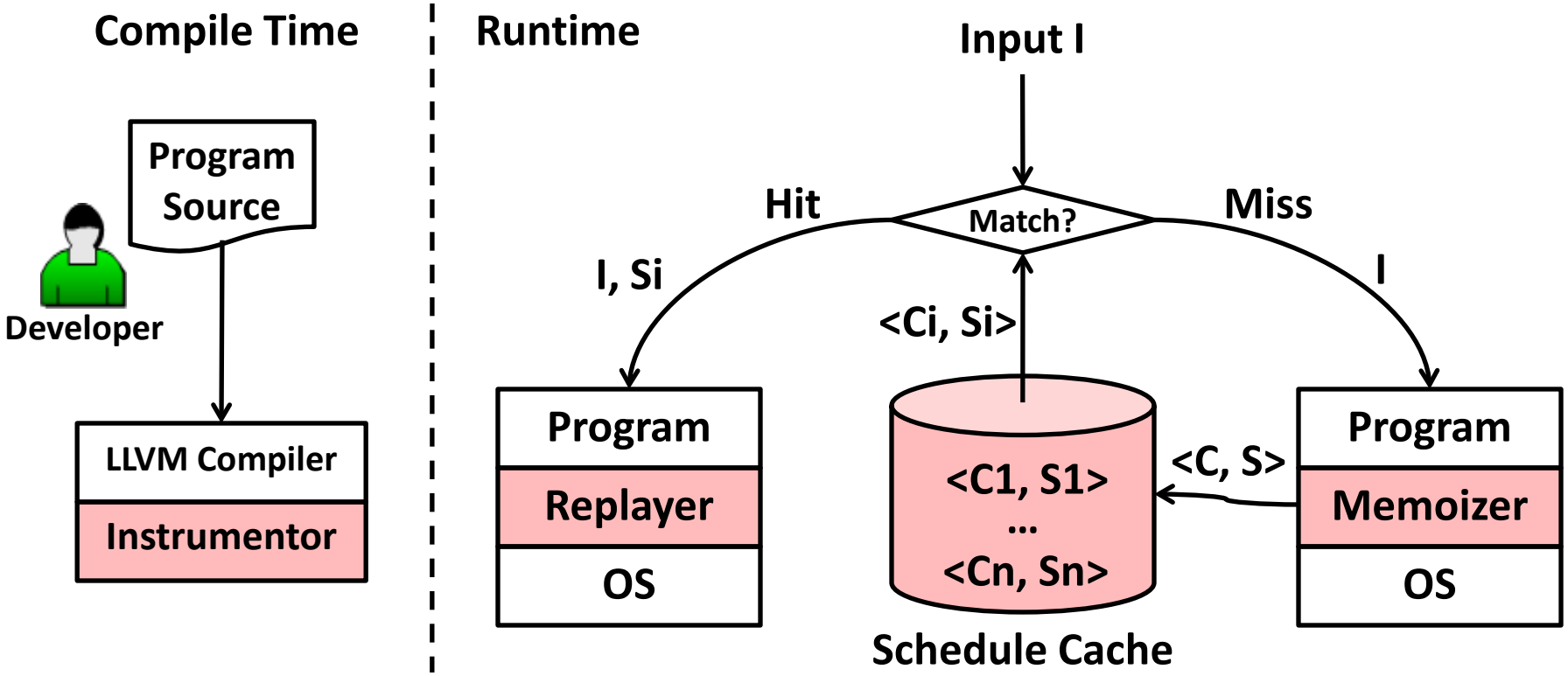
Summary of Results

- Evaluated on diverse set of 14 programs
  - Apache, MySQL, PBZip2, 11 scientific programs
  - Real and synthetic workloads
- Easy to use: < 10 lines for 13 out of 14
- Stable: e.g., 100 schedules to process over 90% of real HTTP trace with 122K requests
- Reasonable overhead: < 10% for 9 out of 14

Outline

- TERN overview
- An Example
- Evaluation
- Conclusion

Overview of TERN



TERN components are shaded

Outline

- TERN overview
- An Example
- Evaluation
- Conclusion

Simplified PBZip2 Code

```
main(int argc, char *argv[]) {
  int i;
  int nthread = argv[1];        // read input
  int nblock = argv[2];

  for(i=0; i<nthread; ++i)      // create worker threads
    pthread_create(worker);

  for(i=0; i<nblock; ++i) {
    block = bread(i,argv[3]); // read i'th file block
    add(worklist, block);     // add block to work list
  }
}
worker() {
  for(;;) {                     // worker thread code
    block = get(worklist);      // get a block from work list
    compress(block);            // compress block
  }
}
```

Annotating Source

```
main(int argc, char *argv[]) {
  int i;
  int nthread = argv[1];
  int nblock = argv[2];
  symbolic(&nthread);  // marking inputs affecting schedule
  for(i=0; i<nthread; ++i)
    pthread_create(worker);  // TERN intercepts
  symbolic(&nblock);  // marking inputs affecting schedule
  for(i=0; i<nblock; ++i) {
    block = bread(i,argv[3]);
    add(worklist, block);  // TERN intercepts
  }
}
worker() {
  for(;;) {
    block = get(worklist); // TERN intercepts
    compress(block);
  }
}
// TERN tolerates inaccuracy in annotations.
```

Memoizing Schedules

```
cmd$ pbzip2 2 2 foo.txt
```

```
main(int argc, char *argv[]) {
  int i;
T1 int nthread = argv[1];// 2
  int nblock = argv[2]; // 2
T1 symbolic(&nthread);
T1 for(i=0; i<nthread; ++i)
T1   pthread_create(worker);
T1 symbolic(&nblock);
T1 for(i=0; i<nblock; ++i) {
     block = bread(i,argv[3]);
T1   add(worklist, block);
T1 }
}
worker() {
T2 T3 for(;;) {
T2 T3   block = get(worklist);
T2 T3   compress(block);
  }
}
```

Synchronization order

T1	T2	T3
p…create		
p…create		
add		
	get	
add		
		get

Constraints

0 < nthread	? true
1 < nthread	? true
2 < nthread	? false
0 < nblock	? true
1 < nblock	? true
2 < nblock	? false

Simplifying Constraints

```
main(int argc, char *argv[]) {
  int i;
  int nthread = argv[1];
  int nblock = argv[2];
  symbolic(&nthread);
  for(i=0; i<nthread; ++i)
    pthread_create(worker);
  symbolic(&nblock);
  for(i=0; i<nblock; ++i) {
    block = bread(i,argv[3]);
    add(worklist, block);
  }
}
worker() {
  for(;;) {
    block = get(worklist);
    compress(block);
  }
}
```

```
cmd$ pbzip2 2 2 foo.txt
```

Synchronization order

T1	T2	T3
p…create		
p…create		
add		
	get	
add		
		get

Constraints

```
2 == nthread
2 == nblock
```

Constraint simplification techniques in paper

Reusing Schedules

```
cmd$ pbzip2 2 2 bar.txt
```

```
main(int argc, char *argv[]) {
  int i;
  int nthread = argv[1];// 2
  int nblock = argv[2]; // 2
  symbolic(&nthread);
  for(i=0; i<nthread; ++i)
    pthread_create(worker);
  symbolic(&nblock);
  for(i=0; i<nblock; ++i) {
    block = bread(i,argv[3]);
    add(worklist, block);
  }
}
worker() {
  for(;;) {
    block = get(worklist);
    compress(block);
  }
}
```

Synchronization order

T1	T2	T3
p…create		
p…create		
add		
	get	
add		
		get

Constraints

```
2 == nthread
2 == nblock
```

Outline

- TERN Overview
- An Example
- Evaluation
- Conclusion

Stability Experiment Setup

- Program – Workload
  - **Apache-CS**: 4-day Columbia CS web trace, 122K
  - **MySql-SysBench-simple**: 200K random select queries
  - **MySql-SysBench-tx**: 200K random select, update, insert, and delete queries
  - **PBZip2-usr**: random 10,000 files from "/usr"
- Machine: typical 2.66GHz quad-core Intel
- Methodology
  - Memoize schedules on random 1% to 3% of workload
  - Measure reuse rates on entire workload (**Many → 1**)
    - Reuse rate: % of inputs processed with memoized schedules

How Often Can TERN Reuse Schedules?

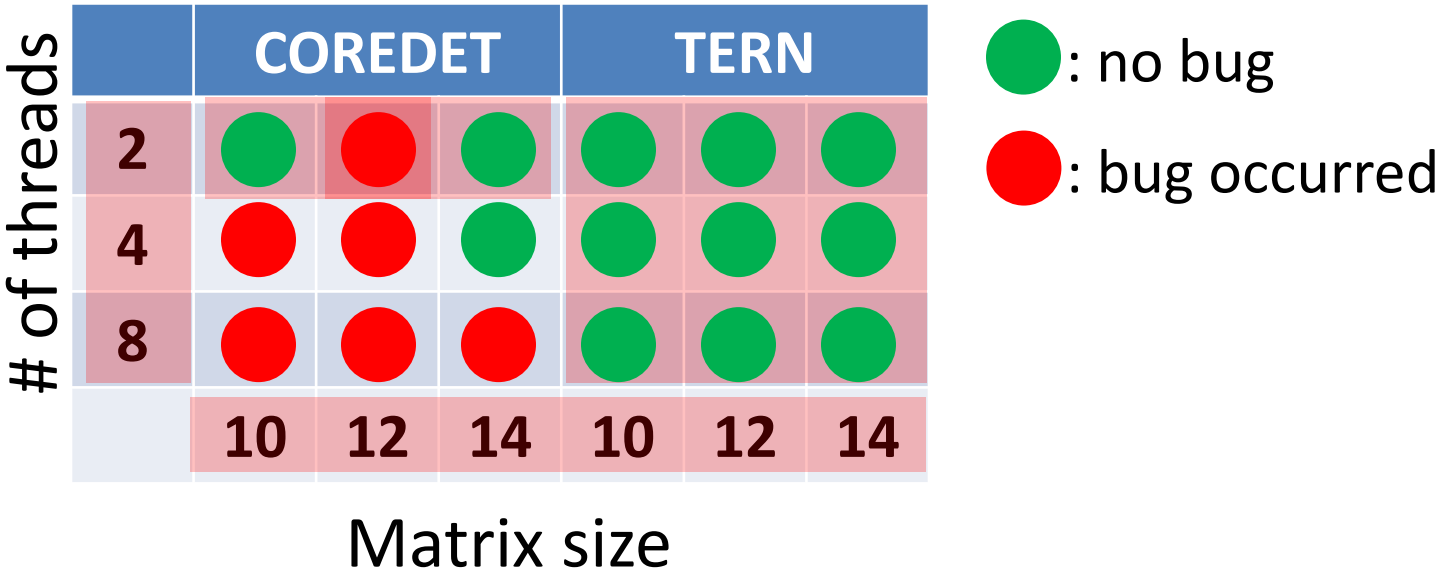
Program-Workload	Reuse Rate (%)	# Schedules
Apache-CS	**90.3**	**100**
MySQL-SysBench-Simple	**94.0**	**50**
MySQL-SysBench-tx	**44.2**	**109**
PBZip2-usr	**96.2**	**90**

- Over 90% reuse rate for three
- Relatively lower reuse rate for MySql-SysBench-tx due to random query types and parameters

Bug Stability Experiment Setup

- Bug stability: when input varies slightly, do bugs occur in one run but disappear in another?
- Compared against COREDET [ASPLOS'10]
  - Open-source, software-only
  - Typical DMT algorithms (one used in dOS)
- Buggy programs: fft, lu, and barnes (SPLASH2)
  - Global variables are printed before assigned correct value
- Methodology: vary thread count and computation amount, then record bug occurrence over 100 runs for COREDET and TERN

Is Buggy Behavior Stable? (fft)



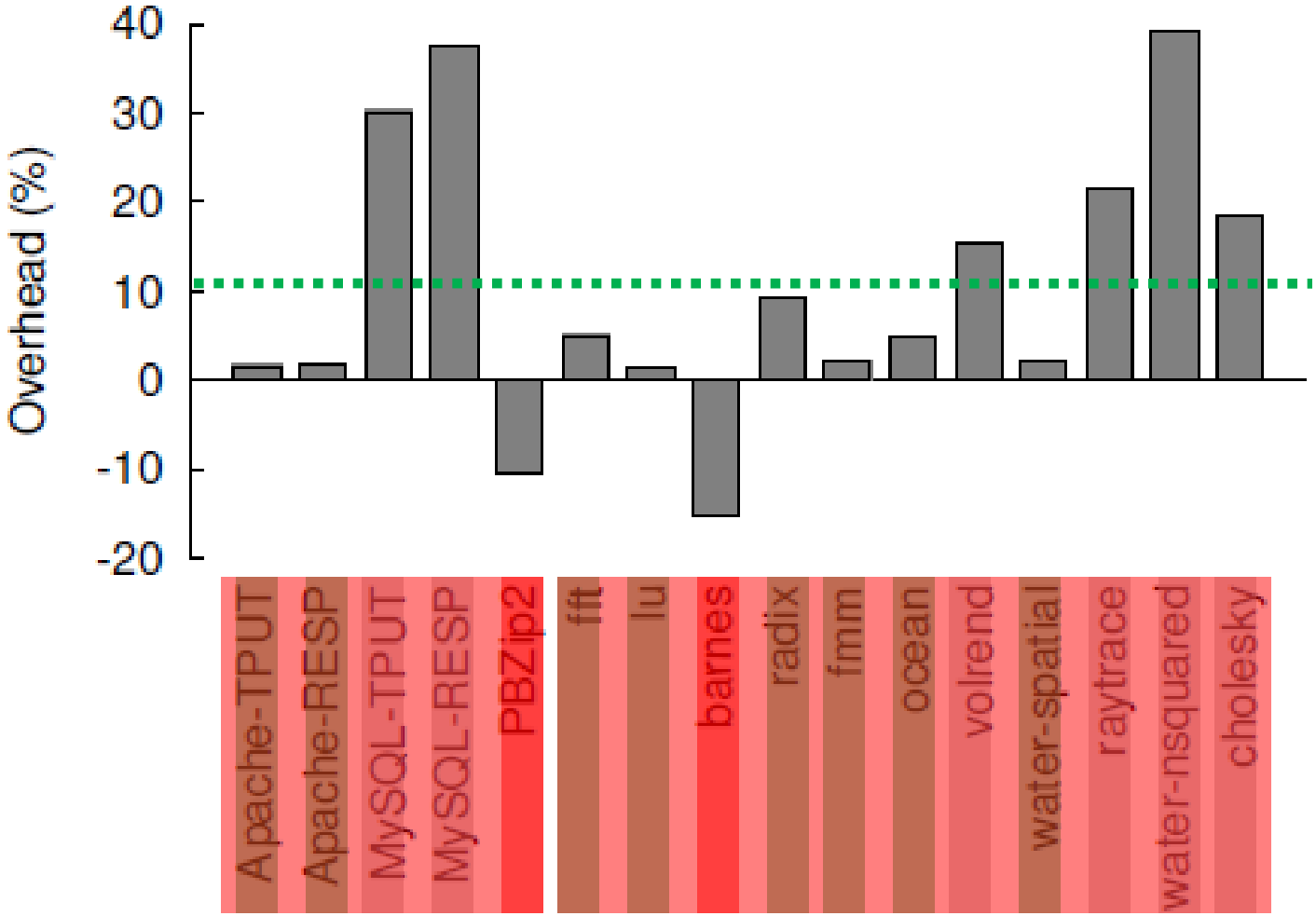
COREDET: 9 schedules, one for each cell.

TERN: only 3 schedules, one for each thread count.

Fewer schedules → lower chance to hit bug → more stable

Similar results for 2 to 64 threads, 2 to 20 matrix size, and the other two buggy programs lu and barnes

Does TERN Incur High Overhead in reuse runs?



Smaller is better. Negative values mean speed up.

Conclusion and Future Work

- Schedule memoization: reuse schedules across different inputs (**Many → 1**)
- TERN: easy to use, stable, deterministic, and fast

- Future work
  - Fast & Deterministic Replay/Replication