TERN:
Stable Deterministic Multithreading through Schedule Memoization

Heming Cui, Jingyue Wu, Chia-che Tsai,
Junfeng Yang

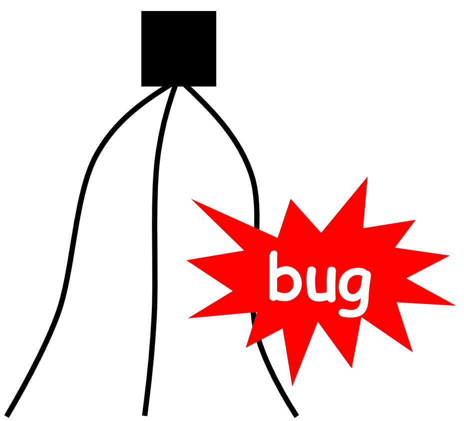
Columbia University

Appeared in OSDI '10

Nondeterministic Execution

- One input ➔ many schedules
- Problem: different runs may show different behaviors, even on the same input

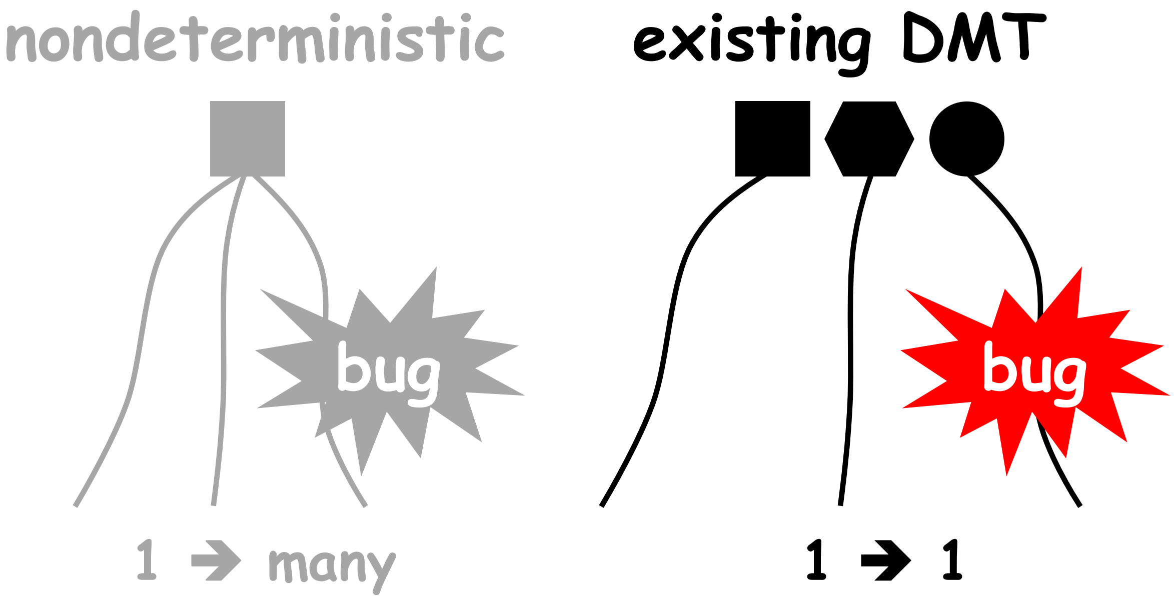
nondeterministic



1 ➔ many

Deterministic Multithreading (DMT)

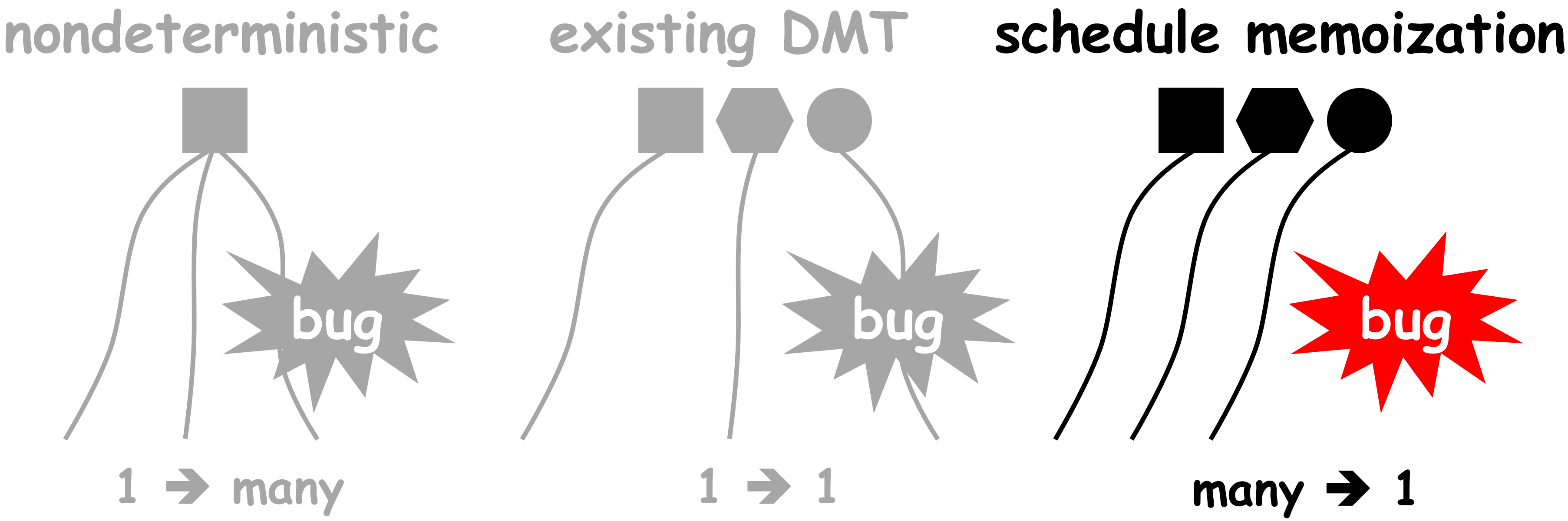
- One input → one schedule
  - [DMP ASPLOS '09], [KENDO ASPLOS '09], [COREDET ASPLOS '10]
- Prob: minor input change → very diff schedule
  - E.g., parallel compression



Confirmed in experiments

Schedule Memoization

- Many inputs → one schedule
  - Memoize schedules and reuse them on future inputs
- Stable: repeat familiar schedules
  - Major benefit: avoid bugs in unknown schedules



Confirmed in experiments

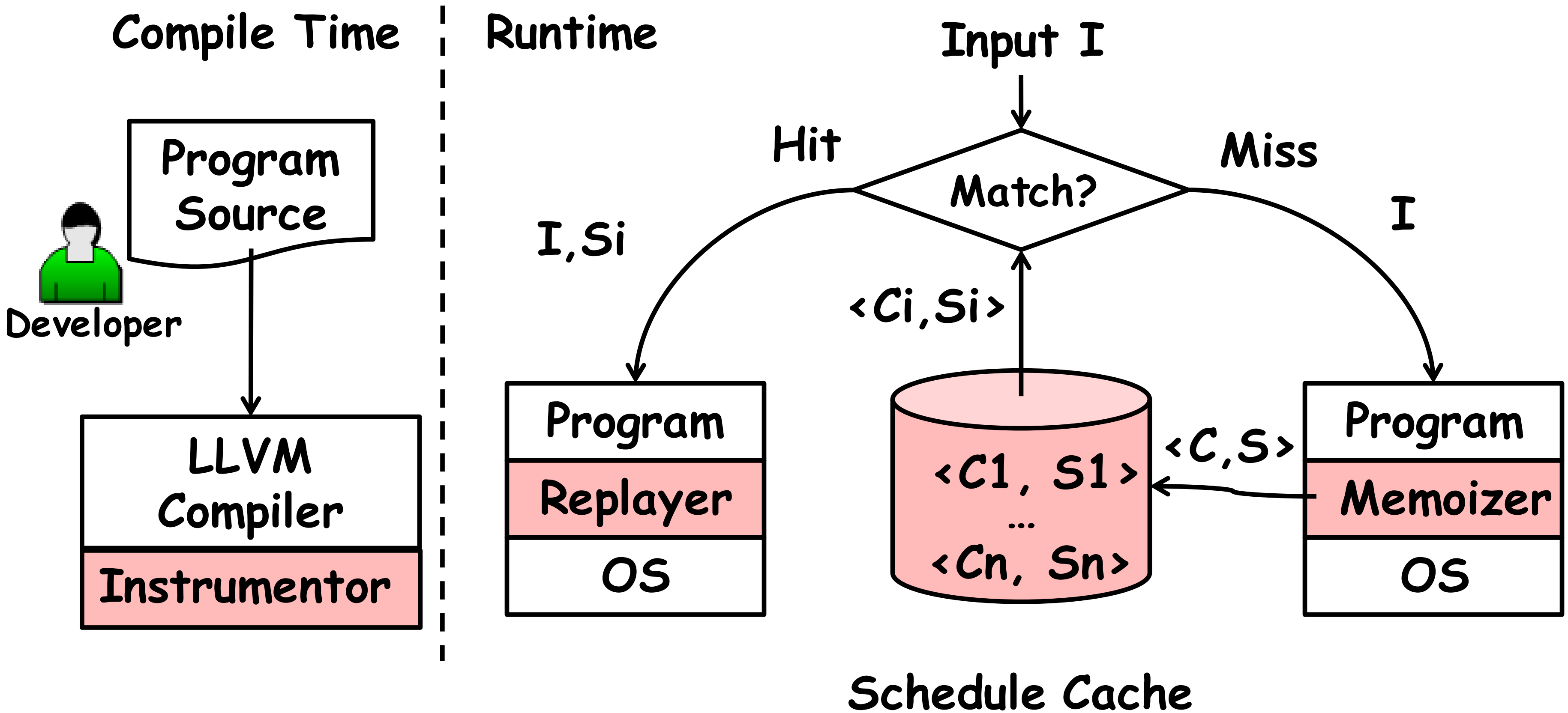
TERN: the First Stable DMT System

- Run on Linux as user-space schedulers
- To memoize a new schedule
  - Memoize total order of synch op 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
- Evaluated on 14 prog, real & synthetic workloads
  - Apache, MySQL, PBZip2, 11 scientific programs
  - Results: easy, deterministic, stable, reasonable overhead

Outline

- Overview
- Implementation
- Evaluation
- Conclusion and future work

Overview of TERN



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)
    pthread_create(worker);           // create worker threads

  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 block from work list
    compress(block);                  // compress block
  }
}
```

Annotating Source

```
main(int argc, char *argv[]) {
  int i;
  int nthread = argv[1];        // TERN tolerates inaccuracy
  int nblock = argv[2];         in annotations
  symbolic(&nthread);          // mark input affecting schedule
  for(i=0; i<nthread; ++i)
    pthread_create(worker);    // TERN intercepts
  symbolic(&nblock);           // mark input 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);
  }
}
```

Memoizing Schedules

```
   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() {
     for(;;) {
       block = get(worklist);
T2 T3  compress(block);
     }
   }
```

cmd> pbzip2 2 2 foo.txt

Synchronization order

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

Redundant constraints

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
```

Reusing Schedules

```
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);
  }
}
```

```
cmd> pbzip2 2 2 bar.txt
```

Synchronization order

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

Constraints

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

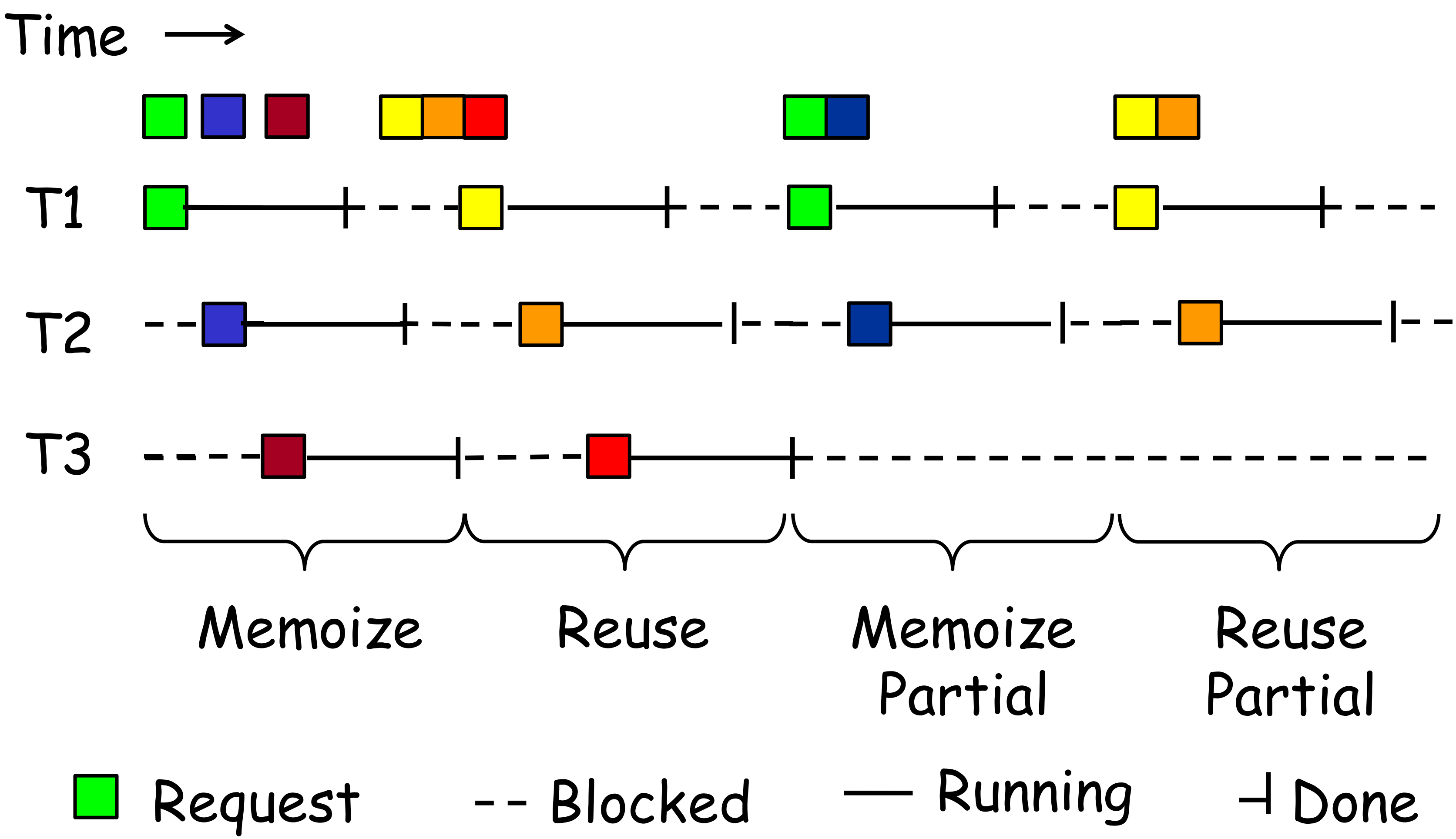
Outline

- Overview
- Implementation
  - Handling server programs
  - Memoizing race-free schedules
- Evaluation
- Conclusion and future work

Handling Server Programs

- Challenges
  - Run continuously → infinite schedule
  - Nondeterministic input timing
- Observation
  - Servers tend to go back to same quiescent states
- Solution: **windowing**
  - Split continuous request stream into windows
  - Process each window as if server were batch prog
  - Memoize and reuse at windows granularity

Windowing in Action



- Reasonable overhead

Outline

- Overview
- Implementation
  - Handling server programs
  - Memoizing race-free schedules
- Evaluation
- Conclusion and future work

Data Races → Nondeterminism

- Race-free runs: synch order = deterministic
  - [RecPlay TOCS '99]
- Racy runs: nondeterministic reuses

```
x = 0;//initialization
```

```
// T1       // T2
lock(L1);
            if(x)
            {
                ...
            }
x = 1;
            lock(L2);
```

Reuse run 1

```
// T1       // T2
lock(L1);
x = 1;
            if(x)
            {
                ...
            }
            lock(L2);
```

Reuse run 2

nondeterministic!

Mitigating Data Races

- Most runs are race free → detect data races, memoize again with diff scheduling algorithm
  - Custom happens-before race detector that flags races not prevented by memoized schedule
  - Two scheduling algorithms: round-robin, run-as-is

```
// T1         // T2
lock(L1);
              if(x)
              {
                  …
              }
x = 1;
              lock(L2);
```

Memoize: race detected

```
// T1         // T2
              if(x)
              {
                  …
              }
              lock(L2);
lock(L1);
x = 1;
```

Memoize again: no race!

Avoiding Symbolic Races

```
// T1          // T2
lock(L1);
               if(a[0])
               {
                   …
               }
a[i] = 1;
               lock(L2);
```

i is symbolic

Memoize: i != 0

Reuse: i == 0

nondeterministic!

- Symbolic race: data-dependent races
- Detect: query constraint solver for "&a[i]==&a[0]"
- Avoid: constrain input s.t. "&a[i]!=&a[0]" ("i!=0")
- Results: race-free schedules for 12 of 14 prog

Limitations

- Best-effort determinism
  - Ongoing: full determinism with efficiency, leveraging [LOOM OSDI '10]
- Manual Annotation
  - Ongoing: automatically identify input data affecting schedules
- Overhead
  - Experiments show reasonable overhead
  - Similar to many other DMT systems
- Applicability: not for every program/workload
  - Apps may want nondeterminism, rarely reuse schedules, be latency sensitive, have intractable constraints, …

Outline

- Overview
- Implementation
  - Handling server programs
  - Memoizing race-free schedules
- Evaluation
- Conclusion and future work

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"

- Methodology
  - Memoize schedules on random 1% to 3% of workload
  - Measure reuse rates on entire workload (**Many → 1**)
    - Reuse rate: input % processed with memoized schedules

How Often Can TERN Reuse Schedules?

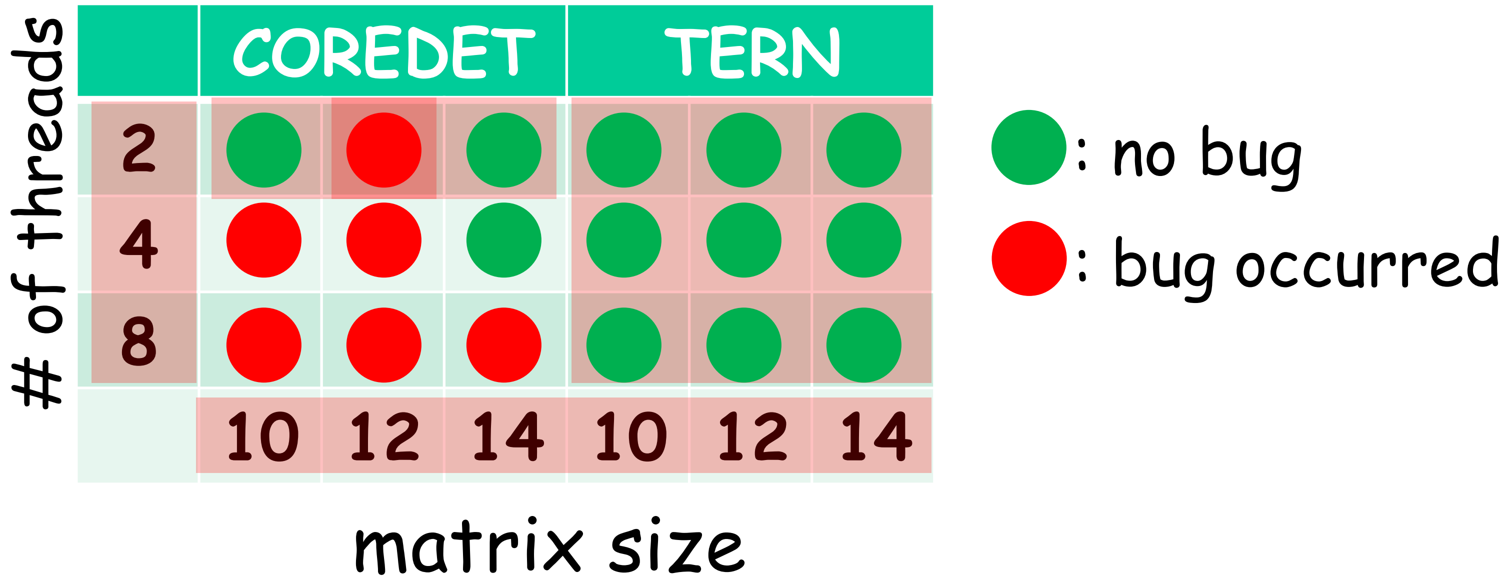
Program-Workload	Reuse Rate (%)	# Schedules
Apache-CS	90.3	100
MySQL-SysBench-Simple	94.0	50
MySQL-SysBench-tx	44.2	109
PBZip2-usr	96.2	90

- 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
- Buggy prog: fft, lu, and barnes (SPLASH2)
  - Global variables used before assigned
- Methodology: vary thread count and computation amount, then record bug occurrence over 100 runs

Is Buggy Behavior Stable? (fft)



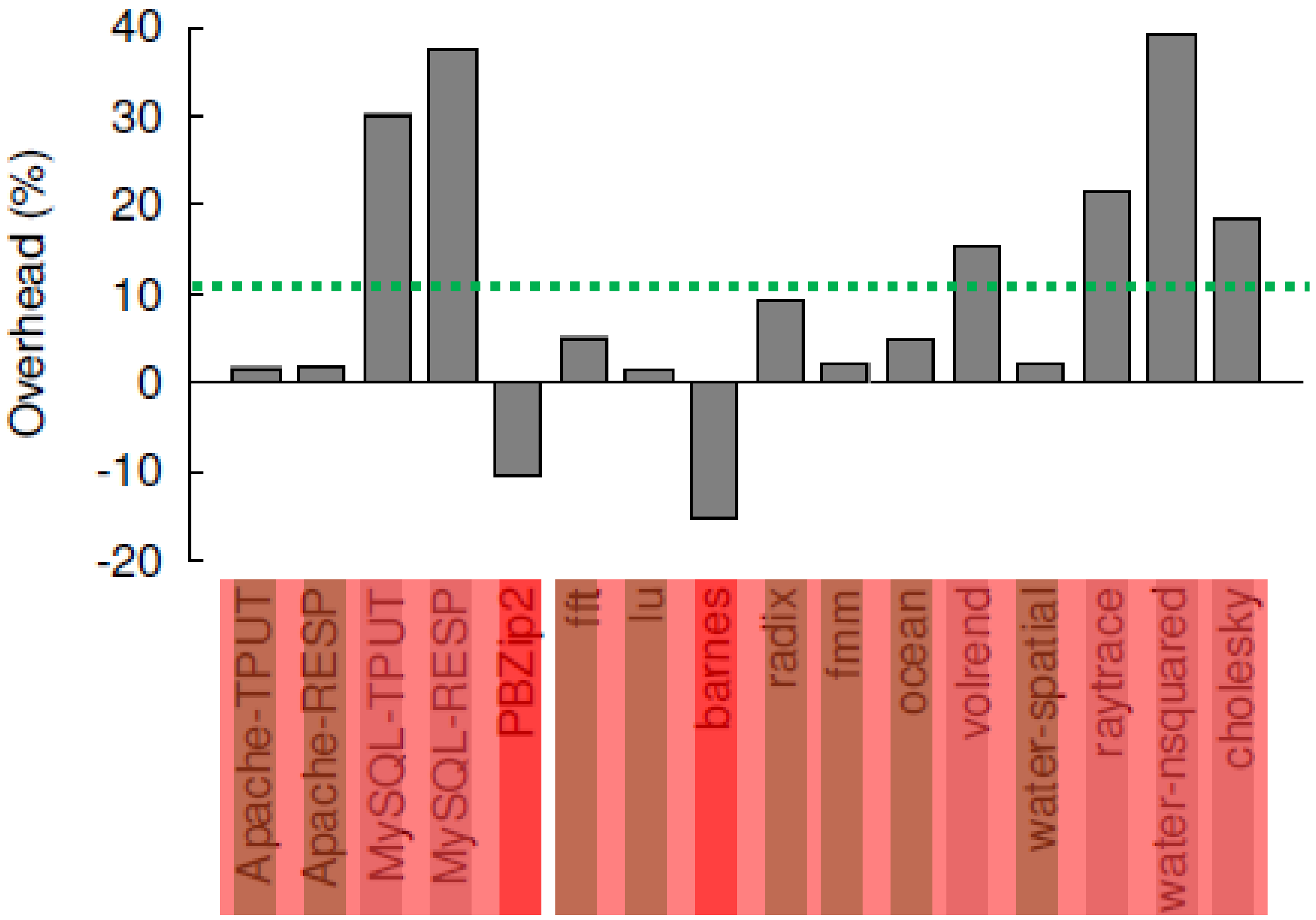
COREDET: 9 schedules, one for each cell.

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

Fewer schedules → more stable

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

Overhead in Reuse Runs?



Smaller Y = better; negative = speedup.

Related Work

- Deterministic Execution
  - [Grace OOPSLA '09], [Kendo ASPLOS '09], [DMP ASPLOS '09], [COREDET ASPLOS '10], [dOS OSDI '10] [Determinator OSDI '10]
- Deterministic Replay
  - [ReVirt OSDI '02] , [SMP-ReVirt VEE '08], [Capo ASPLOS '09], [PRES SOSP '09], [ODR SOSP '09], [Scribe SIGMETRICS '10]
- Symbolic Execution
  - [CUTE FSE-13], [EXE CCS '06], [Yang et el SP '06], [Bouncer SOSP '07], [KLEE OSDI '08], [Castro et el ASPLOS '08]
- Concurrency Errors
  - [Eraser TOCS '97], [Racex SOSP '03], [RaceTrack SOSP '05], [Avio ASPLOS '06], [Lu et el ASPLOS '08], [CTrigger ASPLOS '09]

Future Work

- Address TERN limitations
  - Automatic, fully deterministic, and fast
- Broaden TERN scope
  - System-wide schedule memoization: kernel, multiple processes, distributed
  - Other types of concurrency errors: deadlocks, atomicity errors, …
- Build TERN applications
  - Fast & deterministic replay on multiprocessor
  - Fast & deterministic replication on multiprocessor
  - Better verification of multithreaded programs

Conclusion

- Schedule memoization: **many → 1**
  - Reuse schedules across similar inputs
- TERN: the first stable DMT
  - Runs as user-space schedulers
  - Works on server programs: windowing
- Results: easy to use, more stable and deterministic, reasonable overhead