DeTrans: Deterministic and Parallel Execution of Transactions

Vesna Smiljković, Srđan Stipić, Osman Unsal, Adrián Cristal, Mateo Valero
Barcelona Supercomputing Center, Universitat Politècnica de Catalunya, Spain

Christof Fetzer
Technische Universität Dresden, Germany
Introduction - TM

- **Transactional Memory (TM)**
  - A synchronization mechanism
  - A group of operations for accessing shared memory (loads, stores) is executed atomically, isolated from others and preserving memory consistency
  - Replacement for locks
  - Advantages:
    - programming is simpler and less error-prone
Introduction - TM

➢ **Transactional Memory (TM)**
  ➢ A synchronization mechanism
  ➢ A group of operations for accessing shared memory (loads, stores) is executed atomically, isolated from others and preserving memory consistency
  ➢ Replacement for locks
  ➢ Advantages:
    ➢ programming is simpler and less error-prone
  ➢ **Cannot solve all concurrency bugs**
Introduction - Determinism

- Nondeterminism
  - Multi-threaded programs run nondeterministically
  - Threads interleave in arbitrary order
  - Threads access shared memory locations in different order
  - The same input → a different output
  - A bug appears in one run, it is hidden in another
Introduction - Determinism

- Nondeterminism
  - Multi-threaded programs run nondeterministically
  - Threads interleave in arbitrary order
  - Threads access shared memory locations in different order
  - The same input → a different output
  - A bug appears in one run, it is hidden in another
  - Hard to test and debug ⚠️
Introduction - Determinism

➢ Nondeterminism
   ▶ Multi-threaded programs run nondeterministically
   ▶ Threads interleave in arbitrary order
   ▶ Threads access shared memory locations in different order
   ▶ The same input → a different output
   ▶ A bug appears in one run, it is hidden in another
   ▶ **Hard to test and debug** 🚫

➢ Determinism
   ▶ Repeatability: threads always interleave in the same order
   ▶ The same input → the same output
   ▶ Easier to test and debug: run, test and debug only one interleaving
Introduction - Determinism

➢ **Nondeterminism**
  ▶ Multi-threaded programs run nondeterministically
  ▶ Threads interleave in arbitrary order
  ▶ Threads access shared memory locations in different order
  ▶ The same input → a different output
  ▶ A bug appears in one run, it is hidden in another
  ▶ **Hard to test and debug** 🚫

➢ **Determinism**
  ▶ Repeatability: threads always interleave in the same order
  ▶ The same input → the same output
  ▶ Easier to test and debug: run, test and debug only one interleaving
  ▶ Weak determinism: deterministic execution of critical sections/transactions (sufficient for data race free programs)
  ▶ **Strong determinism**: deterministic execution of whole programs (important for programs with data races)
DeTrans

- Is a run-time library
- Provides strong determinism for TM applications
- Helps in testing and debugging
- Executes
  - non-transactional code serially in round-robin order
  - transactional code in parallel
- Relies on an STM library for correct execution
- Does not need memory protection hardware nor facilities of the OS
- Runs with low additional overhead and faster than a state-of-the-art system
Motivation – Example 1

➢ Multi-threaded applications are hard to test and debug
➢ TM does not avoid all concurrency bugs
Motivation – Example 1

➢ Multi-threaded applications are hard to test and debug
➢ TM does not avoid all concurrency bugs
➢ Concurrency bugs we want to avoid
Motivation – Example 1

- Multi-threaded applications are hard to test and debug
- TM does not avoid all concurrency bugs
- Concurrency bugs we want to avoid

```c
foo1() {
    tm_atomic {
        n = 0;
    }
    n = 2;
}

foo2() {
    tm_atomic {
        a[n++] = 0;
    }
}
```
Motivation – Example 1

➢ Multi-threaded applications are hard to test and debug
➢ TM does not avoid all concurrency bugs
➢ Concurrency bugs we want to avoid

```
thread1
    foo1() { 
        tm_atomic {
            n = 0;
        } 
        n = 2;
    }

thread2
    foo2() { 
        tm_atomic {
            a[n++] = 0;
        }
    }
```
Motivation – Example 1

➢ Multi-threaded applications are hard to test and debug
➢ TM does not avoid all concurrency bugs
➢ Concurrency bugs we want to avoid

```cpp
thread1

```foo1() {
    tm_atomic {
        n = 0;
    }
    n = 2;
}

thread2

```foo2() {
    tm_atomic {
        a[n++] = 0;
    }
}```
Motivation – Example 1

- Multi-threaded applications are hard to test and debug
- TM does not avoid all concurrency bugs
- Concurrency bugs we want to avoid

```cpp
thread1

foo1() {
  tm_atomic {
    n = 0;
  }
  n = 2;
}

thread2

foo2() {
  tm_atomic {
    a[n++] = 0;
  }
}
```
Motivation – Example 1

➢ Multi-threaded applications are hard to test and debug
➢ TM does not avoid all concurrency bugs
➢ Concurrency bugs we want to avoid

```
thread1 ->
foo1() {
  tm_atomic {
    n = 0;
  }
  n = 2;
}

foo2() {
  tm_atomic {
    a[n++] = 0;
  }
}
```

thread2

```
thread2
n = ???
```
Motivation – Example 1

➢ Multi-threaded applications are hard to test and debug
➢ TM does not avoid all concurrency bugs
➢ Concurrency bugs we want to avoid

thread1

```c
foo1() {
    tm_atomic {
        n = 0;
    }
    n = 2;
}
```

thread2

```c
foo2() {
    tm_atomic {
        a[n++] = 0;
    }
}
```

Interl. #1. An access to an uninitialized variable

```
SEGMENTATION FAULT
```

n = ???
Motivation – Example 2

➢ Multi-threaded applications are hard to test and debug
➢ TM does not avoid all concurrency bugs
➢ Concurrency bugs we want to avoid

```
thread1

foo1() {
    tm.atomic {
        n = 0;
    }
    n = 2;
}

thread2

foo2() {
    tm.atomic {
        a[n++] = 0;
    }
}
```
Motivation – Example 2

- Multi-threaded applications are hard to test and debug
- TM does not avoid all concurrency bugs
- Concurrency bugs we want to avoid

```
thread1

foo1() {
    tm_atomic {
        n = 0;
    }
    n = 2;
}

thread2

foo2() {
    tm_atomic {
        a[n++] = 0;
    }
}
```
Motivation – Example 2

➢ Multi-threaded applications are hard to test and debug
➢ TM does not avoid all concurrency bugs
➢ Concurrency bugs we want to avoid

```c
thread1

foo1() {
    tm_atomic {
        n = 0;
    }
    n = 2;
}

thread2

foo2() {
    tm_atomic {
        a[n++] = 0;
    }
}
```
Motivation – Example 2

➢ Multi-threaded applications are hard to test and debug
➢ TM does not avoid all concurrency bugs
➢ Concurrency bugs we want to avoid

```
thread1

foo1() {
    tm_atomic {
        n = 0;
    }
    n = 2;
}

thread2

foo2() {
    tm_atomic {
        a[n++] = 0;
    }
}
```
Motivation – Example 2

➢ Multi-threaded applications are hard to test and debug
➢ TM does not avoid all concurrency bugs
➢ Concurrency bugs we want to avoid

```
thread1
foo1() {
    tm_atomic {
        n = 0;
    }
    n = 2;
}
```

```
thread2
foo2() {
    tm_atomic {
        a[n++] = 0;
    }
}
```
Motivation – Example 2

- Multi-threaded applications are hard to test and debug
- TM does not avoid all concurrency bugs
- Concurrency bugs we want to avoid

```c
thread1

foo1() {
    tm_atomic {
        n = 0;
    }
    n = 2;
}

thread2

foo2() {
    tm_atomic {
        a[n++] = 0;
    }
}

tmp = n  // n=0
```
Motivation – Example 2

➢ Multi-threaded applications are hard to test and debug
➢ TM does not avoid all concurrency bugs
➢ Concurrency bugs we want to avoid

```
thread1

foo1() {
    tm_atomic {
        n = 0;
    }
    n = 2;
}

thread2

foo2() {
    tm_atomic {
        a[n++] = 0;
    }
}

tmp = n  // n=0
```
Motivation – Example 2

➢ Multi-threaded applications are hard to test and debug
➢ TM does not avoid all concurrency bugs
➢ Concurrency bugs we want to avoid

```c
thread1

foo1() {
    tm_atomic {
        n = 0;
    }
    n = 2;
}

thread2

foo2() {
    tm_atomic {
        a[n++] = 0;
    }
}

tmp = n  //n=0
```
Motivation – Example 2

➢ Multi-threaded applications are hard to test and debug
➢ TM does not avoid all concurrency bugs
➢ Concurrency bugs we want to avoid

thread1

```c
foo1() {
    tm_atomic {
        n = 0;
    }
    n = 2;
}
```

thread2

```c
foo2() {
    tm_atomic {
        a[n++] = 0;
    }
}
```

\[
\text{tmp } = \text{tmp+1}
\]
Motivation – Example 2

➢ Multi-threaded applications are hard to test and debug
➢ TM does not avoid all concurrency bugs
➢ Concurrency bugs we want to avoid

```
thread1

foo1() {
    tm_atomic {
        n = 0;
    }
    n = 2;
}

thread2

foo2() {
    tm_atomic {
        a[n++] = 0;
    }
}
```

```
---

n = tmp // n=1
```
Motivation – Example 2

- Multi-threaded applications are hard to test and debug
- TM does not avoid all concurrency bugs
- Concurrency bugs we want to avoid

```c
thread1

foo1() {
    tm_atomic {
        n = 0;
        n = 2;
    }
}

thread2

foo2() {
    tm_atomic {
        a[n++] = 0;
    }
}

n = tmp  // n=1

What happened with n=2???
```
Motivation – Example 2

➢ Multi-threaded applications are hard to test and debug
➢ TM does not avoid all concurrency bugs
➢ Concurrency bugs we want to avoid

thread1

```cpp
foo1() {
    tm_atomic {
        n = 0;
    }
    n = 2;
}
```

thread2

```cpp
foo2() {
    tm_atomic {
        a[n++] = 0;
    }
}
```

What happened with n=2???

Interl. #2: An update of a shared variable out of transactions

MEMORY INCONSISTENCY
Motivation

➢ Multi-threaded applications are hard to test and debug
➢ TM does not avoid all concurrency bugs
➢ Concurrency bugs we want to avoid

foo1() {
    tm_atomic {
        n = 0;
    }
    n = 2;
}

foo2() {
    tm_atomic {
        a[n++] = 0;
    }
}
Motivation

➢ Multi-threaded applications are hard to test and debug
➢ TM does not avoid all concurrency bugs
➢ Concurrency bugs we want to avoid

```c
foo1() {
    tm_atomic {
        n = 0;
    }
    n = 2;
}
```
```c
foo2() {
    tm_atomic {
        a[n++] = 0;
    }
}
```

Other interleaving

The program finishes SUCCESSFULLY
Motivation

➢ Multi-threaded applications are hard to test and debug
➢ TM does not avoid all concurrency bugs
➢ Concurrency bugs we want to avoid

```c
foo1() {
    tm_atomic {
        n = 0;
    }
    n = 2;
}

foo2() {
    tm_atomic {
        a[n++] = 0;
    }
}
```

Interl. #1. An access to an uninitialized variable

⇒ SEGMENTATION FAULT

Interl. #2: An update of a shared variable out of transactions

⇒ MEMORY INCONSISTENCY

Other interleaving

⇒ The program finishes SUCCESSFULLY
Motivation

➢ Multi-threaded applications are hard to test and debug
➢ TM does not avoid all concurrency bugs
➢ Concurrency bugs we want to avoid

Hard to test and debug!

```c
foo1() {
    tm_atomic {
        n = 0;
    }
    n = 2;
}
```

Interl. #1. An access to an uninitialized variable

```
foo2() {
    tm_atomic {
        a[n++] = 0;
    }
}
```

SEGMENTATION FAULT

Interl. #2: An update of a shared variable out of transactions

MEMORY INCONSISTENCY

Other interleaving

The program finishes SUCCESSFULLY
Motivation

- Multi-threaded applications are hard to test and debug
- TM does not avoid all concurrency bugs
- Concurrency bugs we want to avoid

Hard to test and debug!

Determinism: run, test, debug only 1 interl.

```c
foo1() {
    tm_atomic {
        n = 0;
    }
    n = 2;
}
```

```
foo2() {
    tm_atomic {
        a[n++] = 0;
    }
}
```

Interl. #1. An access to an uninitialized variable
SEGMENTATION FAULT

Interl. #2: An update of a shared variable out of transactions
MEMORY INCONSISTENCY

Other interleaving
The program finishes SUCCESSFULLY
Motivation

- Multi-threaded applications are hard to test and debug
- TM does not avoid all concurrency bugs
- Concurrency bugs we want to avoid
  - Hard to test and debug!
  - Determinism: run, test, debug only 1 interl.
    (not a data-race detector!!!)

```c
foo1() {
    tm_atomic {
        n = 0;
    }
    n = 2;
}
```

Interl. #1. An access to an uninitialized variable

```
foo2() {
    tm_atomic {
        a[n++] = 0;
    }
}
```

SEGMENTATION FAULT

Interl. #2: An update of a shared variable out of transactions

```
Other interleaving
```

MEMORY INCONSISTENCY

The program finishes SUCCESSFULLY
Motivation

➢ Multi-threaded applications are hard to test and debug
➢ TM does not avoid all concurrency bugs
➢ Concurrency bugs we want to avoid

Hard to test and debug!

Determinism: run, test, debug only 1 interl. (not a data-race detector!!)

Strong determinism needed!

\begin{verbatim}
foo1() {
    tm_atomic {
        n = 0;
    }
    n = 2;
}

foo2() {
    tm_atomic {
        a[n++] = 0;
    }
}
\end{verbatim}

Interl. #1. An access to an uninitialized variable
\hspace{1.5cm} \Downarrow \hspace{1.5cm} \text{SEGMENTATION FAULT}

Interl. #2: An update of a shared variable out of transactions
\hspace{1.5cm} \Downarrow \hspace{1.5cm} \text{MEMORY INCONSISTENCY}

Other interleaving
\hspace{1.5cm} \Downarrow \hspace{1.5cm} \text{The program finishes SUCCESSFULLY}
Other systems for deterministic execution of TM apps?

- **Dthreads**\[^1\]
  - Threads replaced with processes (each working on a private mem copy)
  - Syscalls and memory protection HW
  - `pthread_*` as synchronization points

State of the art

➢ Other systems for deterministic execution of TM apps?
  ➢ Dthreads\(^1\)
    ➢ Threads replaced with processes (each working on a private mem copy)
    ➢ Syscalls and memory protection HW
    ➢ pthread_* as synchronization points

```c
main() {
  counter = 0;
  pthread_create(thread1, increment);
  pthread_create(thread2, increment);
  pthread_join(thread1);
  pthread_join(thread2);
}

increment() {
  tm_atomic {
    counter++;
  }
}
```

State of the art

➢ Other systems for deterministic execution of TM apps?
  ➢ Dthreads[1]
    ➢ Threads replaced with processes (each working on a private mem copy)
    ➢ Syscalls and memory protection HW
    ➢ pthread_* as synchronization points

```c
main() {
    counter = 0;
    pthread_create(thread1, increment);
    pthread_create(thread2, increment);
    pthread_join(thread1);
    pthread_join(thread2);
}

increment() {
    tm_atomic {
        counter++;
    }
}
```

State of the art

Other systems for deterministic execution of TM apps?

Dthreads\textsuperscript{[1]}

- Threads replaced with processes (each working on a private mem copy)
- Syscalls and memory protection HW
- pthread\_\* as synchronization points

```c
main() {
    counter = 0;
    pthread_create(thread1, increment);
    pthread_create(thread2, increment);
    pthread_join(thread1);
    pthread_join(thread2);
}

increment() {
    tm_atomic {
        counter++;
    }
}
```

State of the art

➢ Other systems for deterministic execution of TM apps?
  ➢ Dthreads\textsuperscript{[1]}
    ➢ Threads replaced with processes (each working on a private mem copy)
    ➢ Syscalls and memory protection HW
    ➢ pthread_* as synchronization points

```c
main() {
    counter = 0;
    pthread_create(thread1, increment);
    pthread_create(thread2, increment);
    pthread_join(thread1);
    pthread_join(thread2);
}

increment() {
    tm_atomic {
        counter++;
    }
}
```

State of the art

➢ Other systems for deterministic execution of TM apps?
  ➢ Dthreads\textsuperscript{[1]}
    ➢ Threads replaced with processes (each working on a private mem copy)
    ➢ Syscalls and memory protection HW
    ➢ pthread\_\* as synchronization points

```
main() {
    counter = 0;
    pthread_create(thread1, increment);
    pthread_create(thread2, increment);
    pthread_join(thread1);
    pthread_join(thread2);
}

ingrement() {
    tm_atomic {
        counter++;
    }
}
```

State of the art

➢ Other systems for deterministic execution of TM apps?

➢ Dthreads\[1\]
  ➢ Threads replaced with processes (each working on a private mem copy)
  ➢ Syscalls and memory protection HW
  ➢ pthread_* as synchronization points

```
main() {
    counter = 0;
    pthread_create(thread1, increment);
    pthread_create(thread2, increment);
    pthread_join(thread1);
    pthread_join(thread2);
}

increment() {
    tm_atomic {
        counter++;
    }
}
```

State of the art

➢ Other systems for deterministic execution of TM apps?

➢ Dthreads\[1\]
  ▶ Threads replaced with processes (each working on a private mem copy)
  ▶ Syscalls and memory protection HW
  ▶ pthread_* as synchronization points

```
main() {
    counter = 0;
    pthread_create(thread1, increment);
    pthread_create(thread2, increment);
    pthread_join(thread1);
    pthread_join(thread2);
}

increment() {
    tm_atomic {
        counter++;
    }
}
```

Other systems for deterministic execution of TM apps?

Dthreads

- Threads replaced with processes (each working on a private mem copy)
- Syscalls and memory protection HW
- pthread_* as synchronization points

```
main() {
    counter = 0;
    pthread_create(thread1, increment);
    pthread_create(thread2, increment);
    pthread_join(thread1);
    pthread_join(thread2);
}

increment() {
    tm_atomic {
        counter++;
    }
}
```

State of the art

➢ Other systems for deterministic execution of TM apps?

➢ Dthreads\[1\]
  ➢ Threads replaced with processes (each working on a private mem copy)
  ➢ Syscalls and memory protection HW
  ➢ pthread_* as synchronization points

```c
main() {
    counter = 0;
    pthread_create(thread1, increment);
    pthread_create(thread2, increment);
    pthread_join(thread1);
    pthread_join(thread2);
}

increment() {
    tm_atomic {
        counter++;
    }
}
```

counter = 1 at the end???

State of the art

➢ Other systems for deterministic execution of TM apps?
  ➢ Dthreads\(^1\)
    ➢ Threads replaced with processes (each working on a private mem copy)
    ➢ Syscalls and memory protection HW
    ➢ pthread\_* as synchronization points

```c
main() {
    counter = 0;
    pthread_create(thread1, increment);
    pthread_create(thread2, increment);
    pthread_join(thread1);
    pthread_join(thread2);
}

increment() {
    tm_atomic {
        counter++;
    }
}
```

TM applications running with Dthreads

MEMORY INCONSISTENCY

---

Implementation

➢ A double-barrier technique

➢ Serial Phase:
  ✓ Only one thread executes in time
  ✓ Threads execute in round-robin

✓ B1: non-txn code finishes before txn code starts (important for strong determ.)

➢ Parallel Phase: Threads execute in parallel

✓ B2: txn code finishes before non-txn code starts (important for strong determ.)
Implementation

➢ A double-barrier technique
  ➢ Serial Phase:
    ➢ Only one thread executes in time
    ➢ Threads execute in round-robin
  ➢ B1: non-txn code finishes before txn code starts
    (important for strong determ.)
  ➢ Parallel Phase: Threads execute in parallel
  ➢ B2: txn code finishes before non-txn code starts
    (important for strong determ.)

➢ DeTrans-lazy:
  ➢ Serial Phase: execution of non-txn code
  ➢ Parallel Phase: execution of txn code
    ➢ TM lib buffers writes to mem
    ➢ TM lib detects conflicts at commit
  ➢ B2a: commit in round-robin order
  ➢ B2b: like B2
Implementation

A double-barrier technique
- Serial Phase:
  - Only one thread executes in time
  - Threads execute in round-robin
  - B1: non-txn code finishes before txn code starts
    (important for strong determ.)
- Parallel Phase: Threads execute in parallel
  - B2: txn code finishes before non-txn code starts
    (important for strong determ.)

DeTrans-lazy:
- Serial Phase: execution of non-txn code
- Parallel Phase: execution of txn code
  - TM lib buffers writes to mem
  - TM lib detects conflicts at commit
  - B2a: commit in round-robin order
  - B2b: like B2

DeTrans-eager... look at the paper, please.
Evaluation

➢ **Environment**
  ➢ 2 Intel Xeon E5405 processors with 4 cores (8 cores in total), 2GHz, 4GiB RAM
  ➢ TinySTM[2] 1.0.5
  ➢ STAMP[3]
    ➢ STAMP for Dthreads: pthread_mutex_lock/unlock instead of txns
  ➢ Running with 1, 2, 4 and 8 threads

➢ **Verification**
  ➢ Racey[4]

➢ **Profiling**
  ➢ Perf

---

Evaluation - Results

➢ DeTrans in comparison to original:
**DeTrans** in comparison to **original**:

- 1.14x, 1.44x, 1.72x, 2.49x slowdown for 1, 2, 4, 8 threads, respectively
DeTrans in comparison to original:
- 1.14x, 1.44x, 1.72x, 2.49x slowdown for 1, 2, 4, 8 threads, respectively

DeTrans in comparison to Dthreads:
Evaluation - Results

➢ **DeTrans** in comparison to **original**:
  ➢ 1.14x, 1.44x, 1.72x, 2.49x slowdown for 1, 2, 4, 8 threads, respectively

➢ **DeTrans** in comparison to **Dthreads**:
  ➢ 1.43x slowdown for 1 thread
  ➢ 3.99x, 3.39x, 2.44x speedup for 2, 4, 8 threads, respectively
  ➢ Performs better in all benchmarks except Kmeans
Evaluation - Breakdown

➢ Execution breakdown for Vacation
➢ Slowdown in comparison to lock-based 1-threaded execution
➢ app + kernel + determ + libpthread + libstm = 100% exec. time

<table>
<thead>
<tr>
<th>Implementation</th>
<th>1 thread</th>
<th>2 threads</th>
<th>4 threads</th>
<th>8 threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>locks</td>
<td>1</td>
<td>1.25</td>
<td>1.38</td>
<td>1.37</td>
</tr>
<tr>
<td>Dthreads</td>
<td>1.26</td>
<td>24.77</td>
<td>24.7</td>
<td>23.37</td>
</tr>
<tr>
<td></td>
<td>kernel 12.9%</td>
<td>kernel+determ 94.6%</td>
<td>kernel+determ 96.9%</td>
<td>kernel+determ 98.2%</td>
</tr>
<tr>
<td>STM-lazy</td>
<td>2.09</td>
<td>1.37</td>
<td>0.95</td>
<td>0.73</td>
</tr>
<tr>
<td></td>
<td>libstm 59.5%</td>
<td>libstm 60.1%</td>
<td>libstm 61.2%</td>
<td>libstm 61.2%</td>
</tr>
<tr>
<td>DeTrans-lazy</td>
<td>2.19</td>
<td>2.1</td>
<td>2.1</td>
<td>3.13</td>
</tr>
<tr>
<td></td>
<td>determ 2.9%</td>
<td>determ 30.6%</td>
<td>determ 60.5%</td>
<td>determ 85.9%</td>
</tr>
</tbody>
</table>
Evaluation - Breakdown

- Execution breakdown for Vacation
- Slowdown in comparison to lock-based 1-threaded execution
- app + kernel + determ + libpthread + libstm = 100% exec. time

<table>
<thead>
<tr>
<th>Implementation</th>
<th>1 thread</th>
<th>2 threads</th>
<th>4 threads</th>
<th>8 threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>locks</td>
<td>1</td>
<td>1.25</td>
<td>1.38</td>
<td>1.37</td>
</tr>
<tr>
<td>Dthreads</td>
<td>1.26</td>
<td>24.77</td>
<td>24.7</td>
<td>23.37</td>
</tr>
<tr>
<td></td>
<td>kernel 12.9%</td>
<td>kernel+ determ 94.6%</td>
<td>kernel+determ 96.9%</td>
<td>kernel+determ 98.2%</td>
</tr>
<tr>
<td>STM-lazy</td>
<td>2.09</td>
<td>1.37</td>
<td>0.95</td>
<td>0.73</td>
</tr>
<tr>
<td></td>
<td>libstm 59.5%</td>
<td>libstm 60.1%</td>
<td>libstm 61.2%</td>
<td>libstm 61.2%</td>
</tr>
<tr>
<td>DeTrans-lazy</td>
<td>2.19</td>
<td>2.1</td>
<td>2.1</td>
<td>3.13</td>
</tr>
<tr>
<td></td>
<td>determ 2.9%</td>
<td>determ 30.6%</td>
<td>determ 60.5%</td>
<td>determ 85.9%</td>
</tr>
</tbody>
</table>
### Evaluation - Breakdown

- Execution breakdown for Vacation
- Slowdown in comparison to lock-based 1-threaded execution
- app + kernel + determ + libpthread + libstm = 100% exec. time

<table>
<thead>
<tr>
<th>Implementation</th>
<th>1 thread</th>
<th>2 threads</th>
<th>4 threads</th>
<th>8 threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>locks</td>
<td>1</td>
<td>1.25</td>
<td>1.38</td>
<td>1.37</td>
</tr>
<tr>
<td>Dthreads</td>
<td>1.26</td>
<td>24.77</td>
<td>24.7</td>
<td>23.37</td>
</tr>
<tr>
<td></td>
<td>kernel 12.9%</td>
<td>kernel+determ 94.6%</td>
<td>kernel+determ 96.9%</td>
<td>kernel+determ 98.2%</td>
</tr>
<tr>
<td>STM-lazy</td>
<td>2.09</td>
<td>1.37</td>
<td>0.95</td>
<td>0.73</td>
</tr>
<tr>
<td></td>
<td>libstm 59.5%</td>
<td>libstm 60.1%</td>
<td>libstm 61.2%</td>
<td>libstm 61.2%</td>
</tr>
<tr>
<td>DeTrans-lazy</td>
<td>2.19</td>
<td>2.1</td>
<td>2.1</td>
<td>3.13</td>
</tr>
<tr>
<td></td>
<td>determ 2.9%</td>
<td>determ 30.6%</td>
<td>determ 60.5%</td>
<td>determ 85.9%</td>
</tr>
</tbody>
</table>
Evaluation - Breakdown

➢ Execution breakdown for Vacation
➢ Slowdown in comparison to lock-based 1-threaded execution
➢ app + kernel + determ + libpthread + libstm = 100% exec. time

<table>
<thead>
<tr>
<th>Implementation</th>
<th>1 thread</th>
<th>2 threads</th>
<th>4 threads</th>
<th>8 threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>locks</td>
<td>1</td>
<td>1.25</td>
<td>1.38</td>
<td>1.37</td>
</tr>
<tr>
<td>Dthreads</td>
<td>1.26 kernel 12.9%</td>
<td>24.77 kernel+determ 94.6%</td>
<td>24.7 kernel+determ 96.9%</td>
<td>23.37 kernel+determ 98.2%</td>
</tr>
<tr>
<td>STM-lazy</td>
<td>2.09 libstm 59.5%</td>
<td>1.37 libstm 60.1%</td>
<td>0.95 libstm 61.2%</td>
<td>0.73 libstm 61.2%</td>
</tr>
<tr>
<td>DeTrans-lazy</td>
<td>2.19 determ 2.9%</td>
<td>2.1 determ 30.6%</td>
<td>2.1 determ 60.5%</td>
<td>3.13 determ 85.9%</td>
</tr>
</tbody>
</table>
Evaluation - Breakdown

➢ Execution breakdown for Vacation
➢ Slowdown in comparison to lock-based 1-threaded execution
➢ app + kernel + determ + libpthread + libstm = 100% exec. time

<table>
<thead>
<tr>
<th>Implementation</th>
<th>1 thread</th>
<th>2 threads</th>
<th>4 threads</th>
<th>8 threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>locks</td>
<td>1</td>
<td>1.25</td>
<td>1.38</td>
<td>1.37</td>
</tr>
<tr>
<td>Dthreads</td>
<td>1.26</td>
<td>24.77</td>
<td>24.7</td>
<td>23.37</td>
</tr>
<tr>
<td></td>
<td>kernel 12.9%</td>
<td>kernel+determ 94.6%</td>
<td>kernel+determ 96.9%</td>
<td>kernel+determ 98.2%</td>
</tr>
<tr>
<td>STM-lazy</td>
<td>2.09</td>
<td>1.37</td>
<td>0.95</td>
<td>0.73</td>
</tr>
<tr>
<td></td>
<td>libstm 59.5%</td>
<td>libstm 60.1%</td>
<td>libstm 61.2%</td>
<td>libstm 61.2%</td>
</tr>
<tr>
<td>DeTrans-lazy</td>
<td>2.19</td>
<td>2.1</td>
<td>2.1</td>
<td>3.13</td>
</tr>
<tr>
<td></td>
<td>determ 2.9%</td>
<td>determ 30.6%</td>
<td>determ 60.5%</td>
<td>determ 85.9%</td>
</tr>
</tbody>
</table>

➢ DeTrans-lazy in comparison with Dthreads
➢ 11.78x, 11.85x, 7.45x speedup for 2, 4, 8 threads, respectively
Conclusion

➢ DeTrans:
  ➢ Provides strong determinism of TM applications
  ➢ Helps testing and debugging
  ➢ Uses STM libraries with different conflict detection policies
  ➢ Does not use memory protection HW nor facilities of OS

➢ Our evaluation of DeTrans:
  ➢ Compares DeTrans with state of the art Dthreads
  ➢ Shows that DeTrans is faster for multi-threaded execution
    (3.99x, 3.39x, 2.44x for 2, 4, 8 threads, respectively)
Thank you for your attention!
Questions?

Vesna Smiljković
vesna.smiljkovic@bsc.es
Overhead of deterministic execution in Dthreads