In-Place Metainfo Support in DeuceSTM

Ricardo J. Dias, Tiago M. Vale and João M. Lourenço

CITI / Universidade Nova de Lisboa

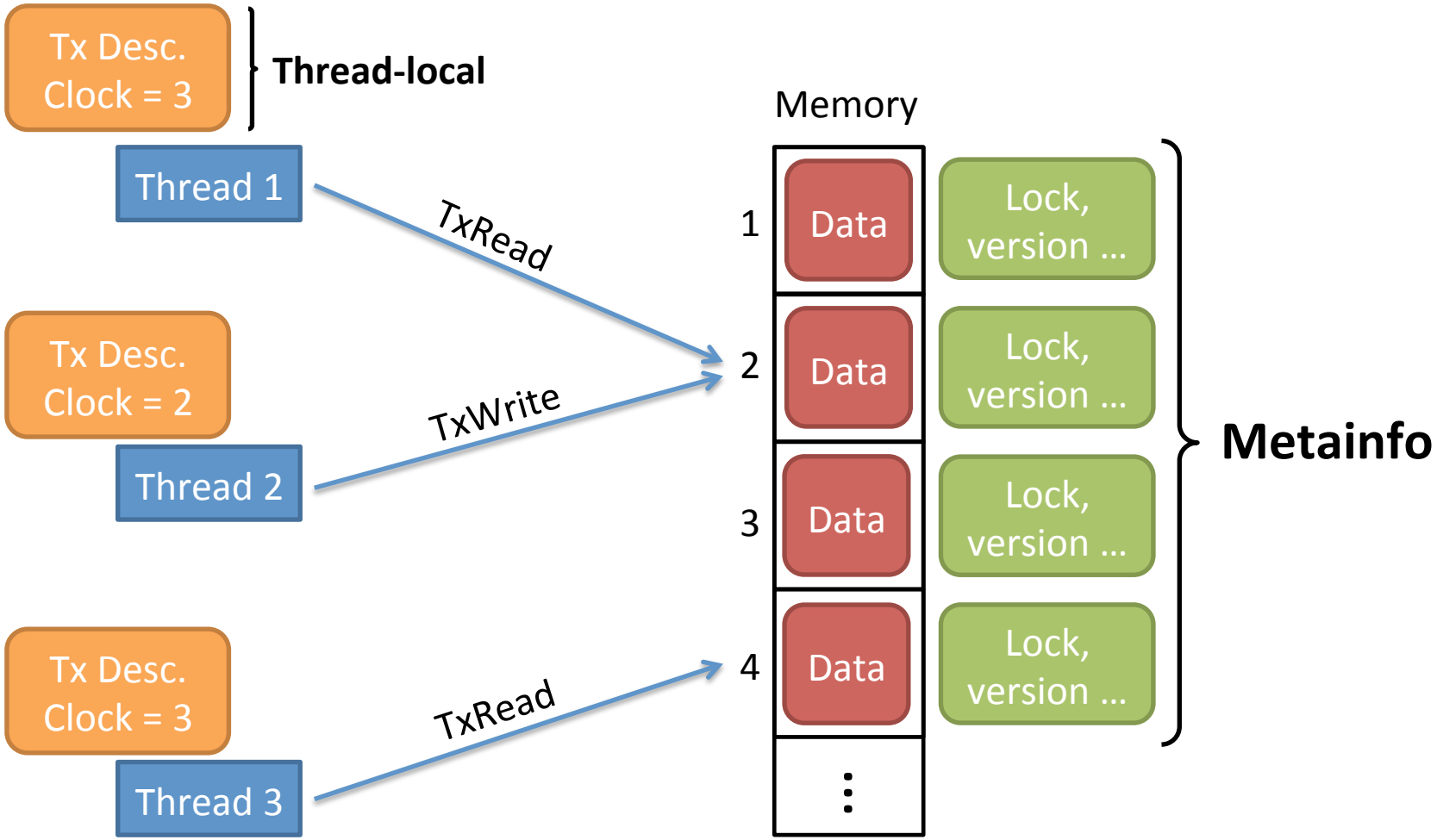
Motivation

- Fair comparison of different STM algorithms
- Require a flexible framework to support different STM implementations
  - Multi-version, lock-based, …
  - Same transactional interface
  - Same benchmarking applications

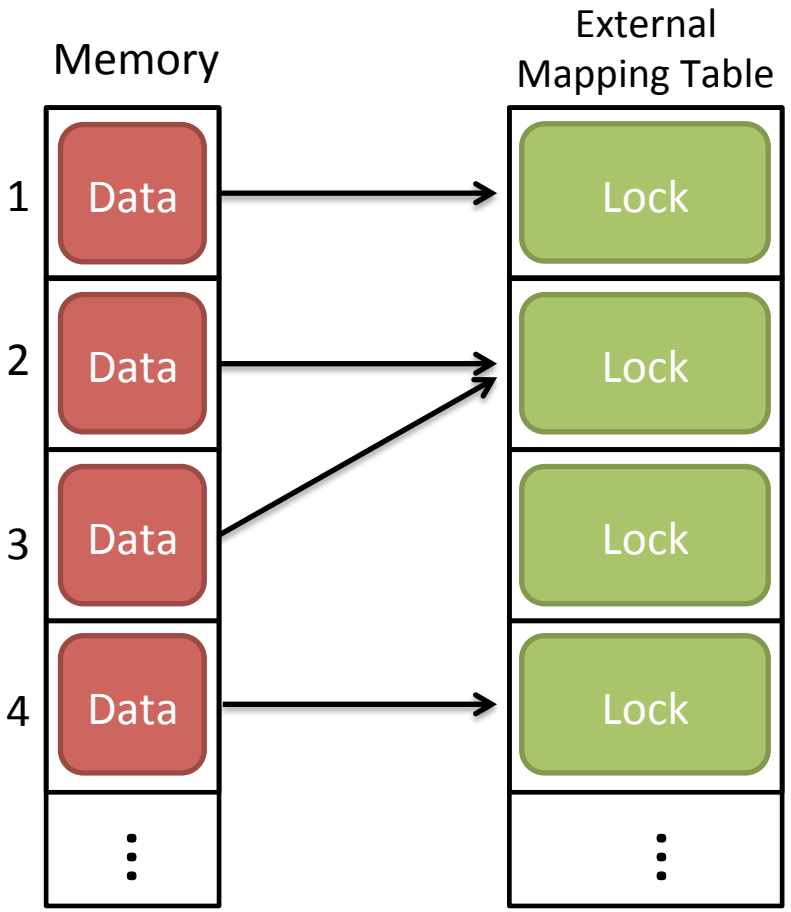
The DeuceSTM

- ☺ Transactions defined by a single Java method annotation: @Atomic
- ☺ Well-defined API for implementing STM algorithms
- ☺ Efficient implementation of some STM algorithms, e.g., TL2, LSA, …
- ☺ Macro-benchmarks available
- ☹ Does not allow the efficient implementation of other STM algorithms, e.g., multi-version
- ☹ Fair comparison is not possible

Transactional Information

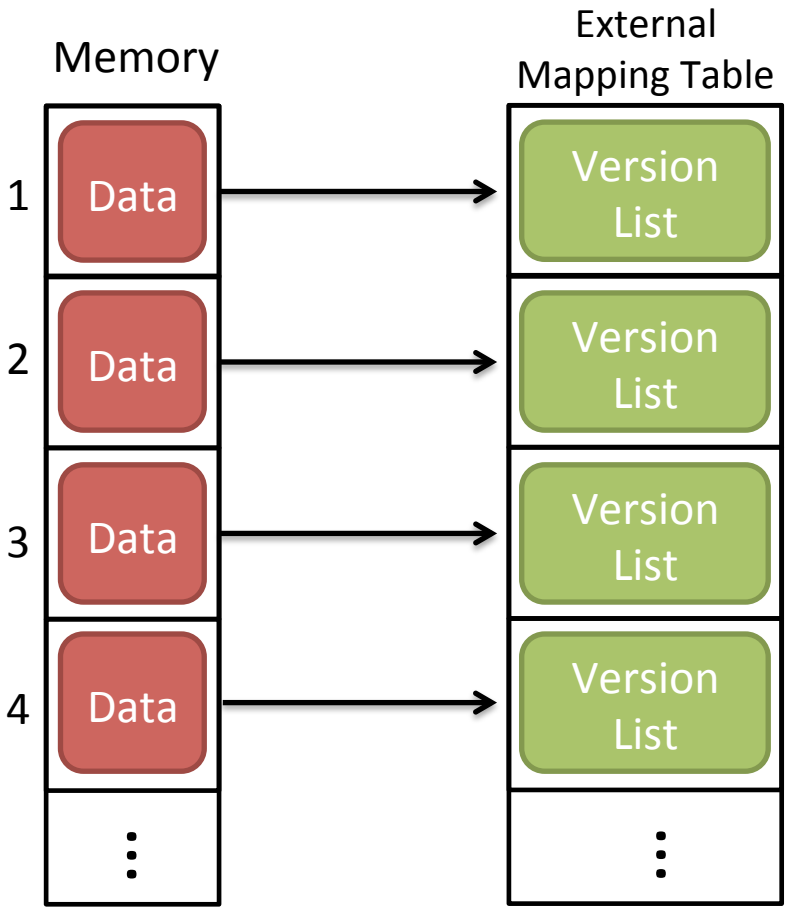


Out-place: N-1



- Efficient implementation using an hash function
- False sharing
- Algorithms:
  - TL2, SwissTM, LSA

Out-place: 1-1



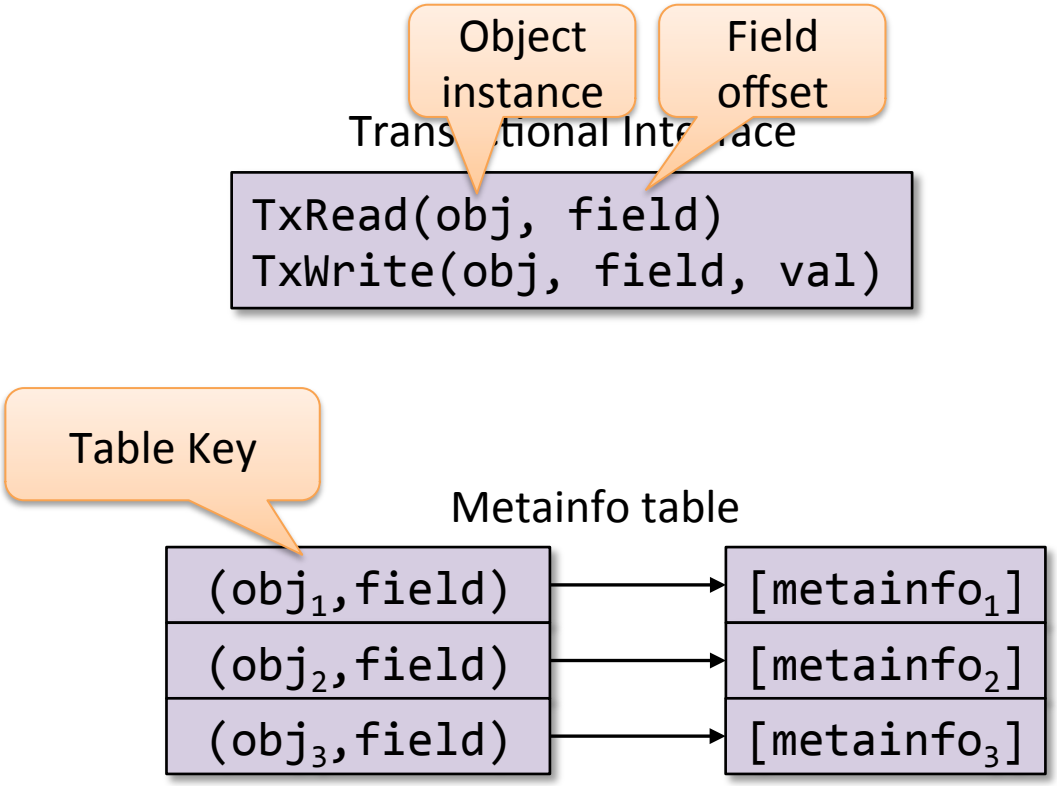
- Hash table with collision list
- Bad performance
- Algorithms:
  - Multi-version algorithms

In-place: 1-1

- Direct access to metainfo using memory reference
- Mostly used in managed
- Algorithms:
  - TL2, SwissTM, LSA, Multi-version algorithms

Add support for In-place in DeuceSTM

Out-place in DeuceSTM



Out-place Instrumentation

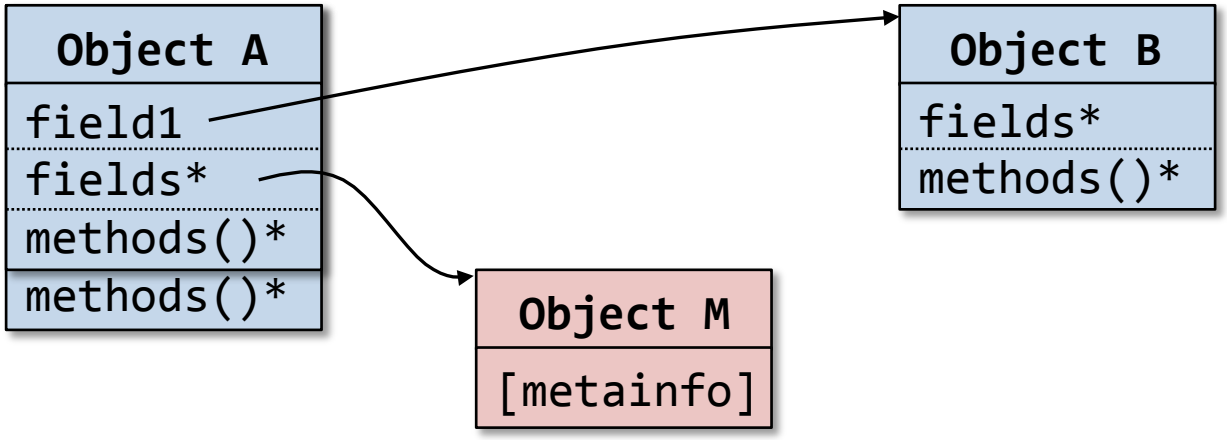
```
class C {
  int x;

  @Atomic foo() {
    x = x+1;
  }
}
```

```
class C {
  int x;
  static int x_off = Offset(x);

  @Atomic foo() {
    int t = TxRead(this, x_off );
    TxWrite(this, x_off , t+1);
  }
}
```

Our In-place Approach



Transactional Interface

```
TxRead(metainfo)
TxWrite(metainfo, val)
```

Our In-place Approach

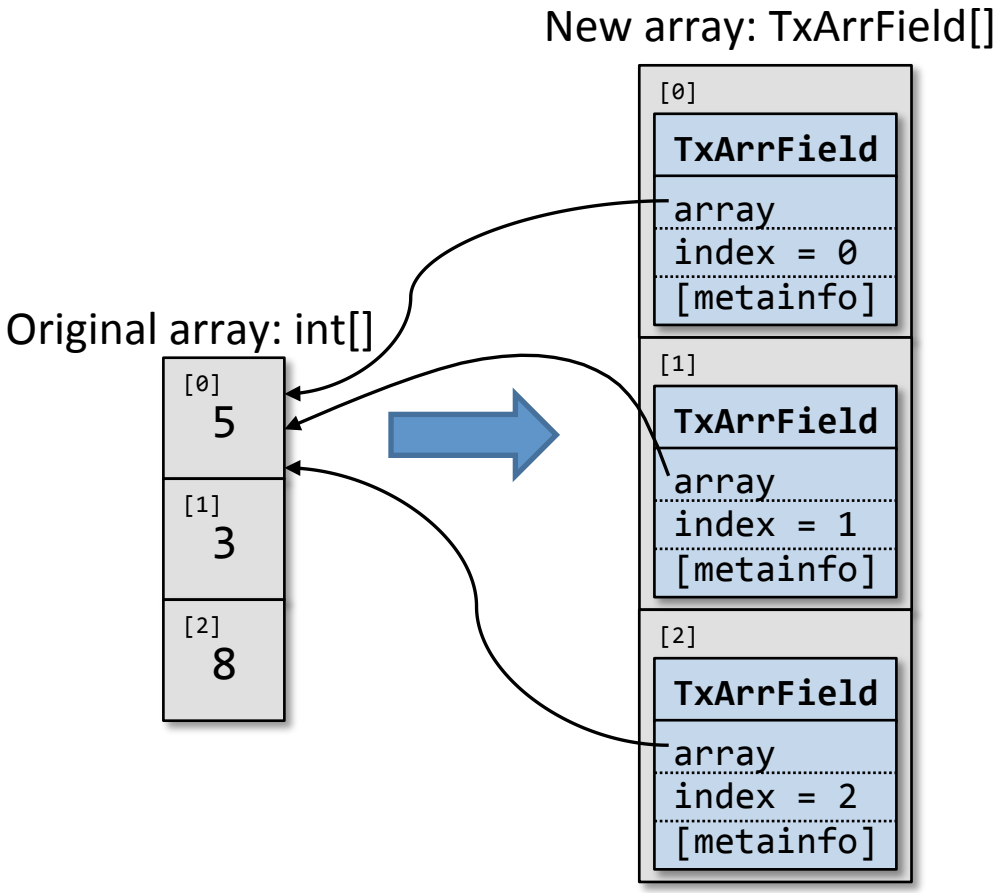
```
class C {
  int x;

  @Atomic foo() {
    x = x+1;
  }
}
```

```
class C {
  int x;
  TxField x_m = new TxField();

  @Atomic foo() {
    int t = TxRead(x_m);
    TxWrite(x_m, t+1);
  }
}
```

Metainfo and arrays



Metainfo and arrays

```
class C {
  int[] a = new int[10];

  @Atomic foo() {
    a[1] = a[2]+1;
  }

  void bar() {
      a[2] = 3;
  }
}
```

```
class C {
  TxArrInt[] a = new TxArrInt[10];
  {
    int[] t = new int[10];
    for (int i=0; i < 10; i++) {
      a[i] = new TxArrInt(i, t);
    }
  }
  TxField a_m = new TxField();

  @Atomic foo() {
    int t = TxRead(a[2]);
    TxWrite(a[1], t+1);
  }

  void bar() {
      a[0].array[3] = 3;
  }
}
```

Experimental Evaluation
Overhead

- Benchmarking algorithm: TL2 using a lock table
- Base case:
  - Using out-place strategy (original DeuceSTM)
- Comparing case:
  - Using in-place strategy
    - Metainfo objects are created for each field
    - We use the metainfo object as the key for the external lock table

Experimental Evaluation

Overhead

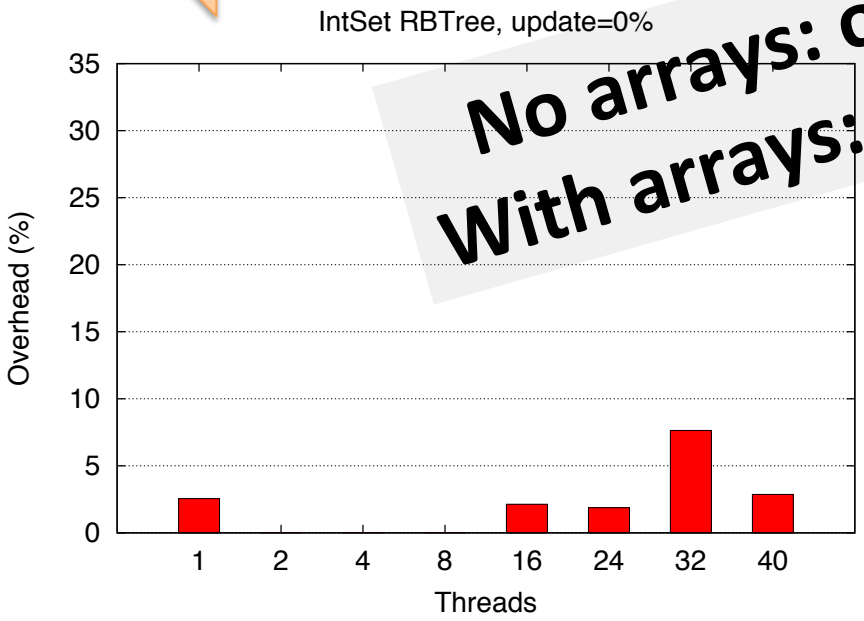
Not using Arrays

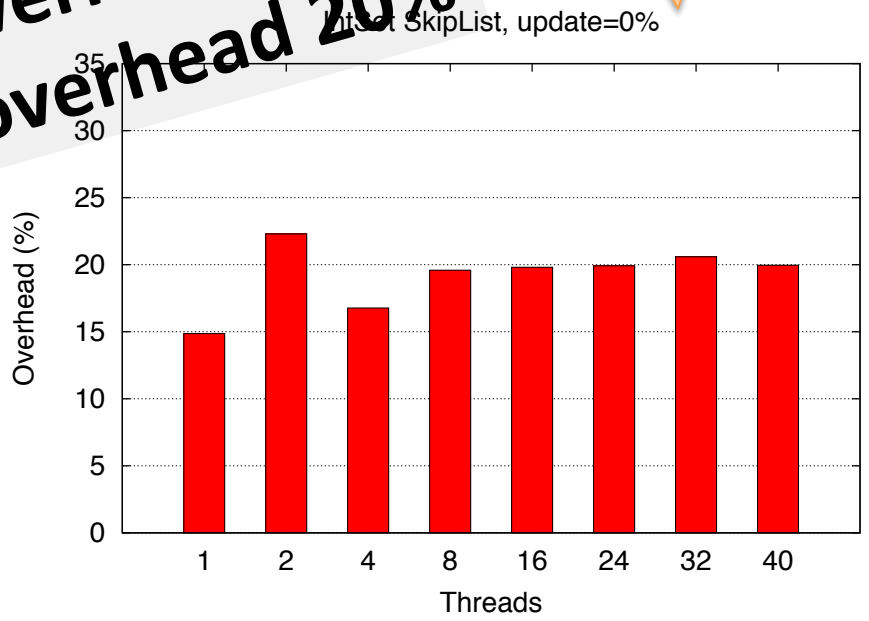
Using Arrays

Write-Update: 0%

No arrays: overhead 3%

With arrays: overhead 20%





Overhead in %

Experimental Evaluation

Overhead

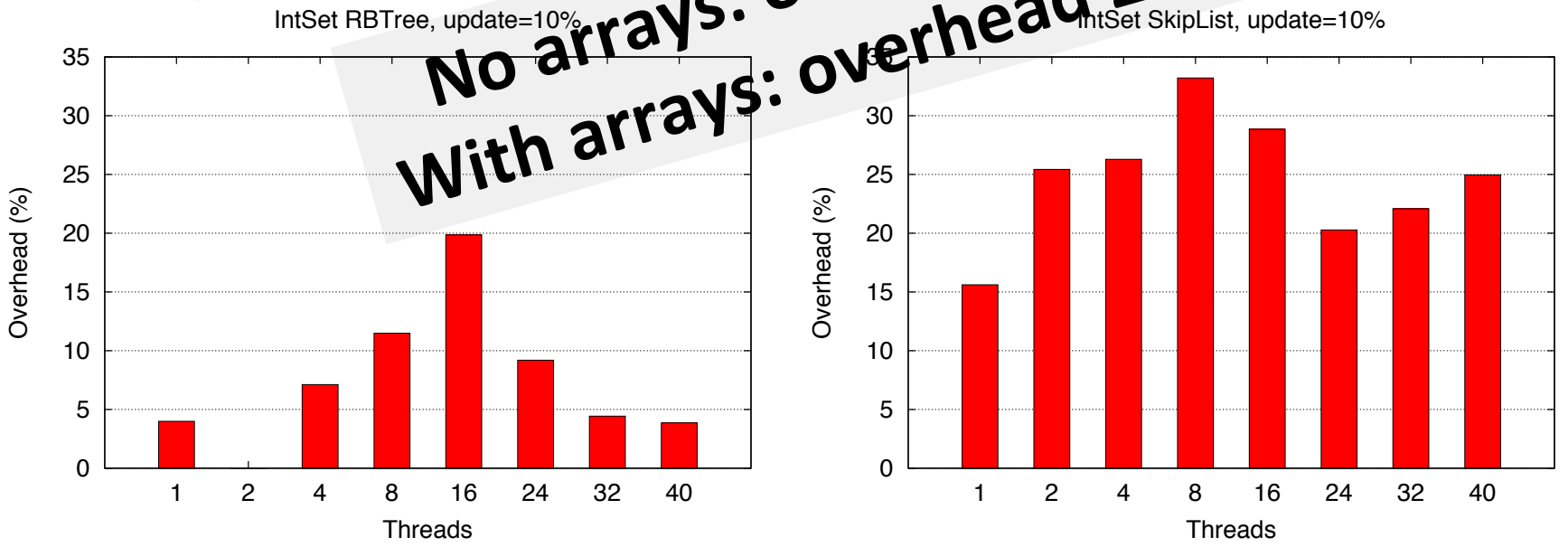
Not using Arrays

Using Arrays

Write-Update: 10%

No arrays: overhead 7%

With arrays: overhead 25%



Overhead in %

Experimental Results

In-place vs. Out-place for Multi-version

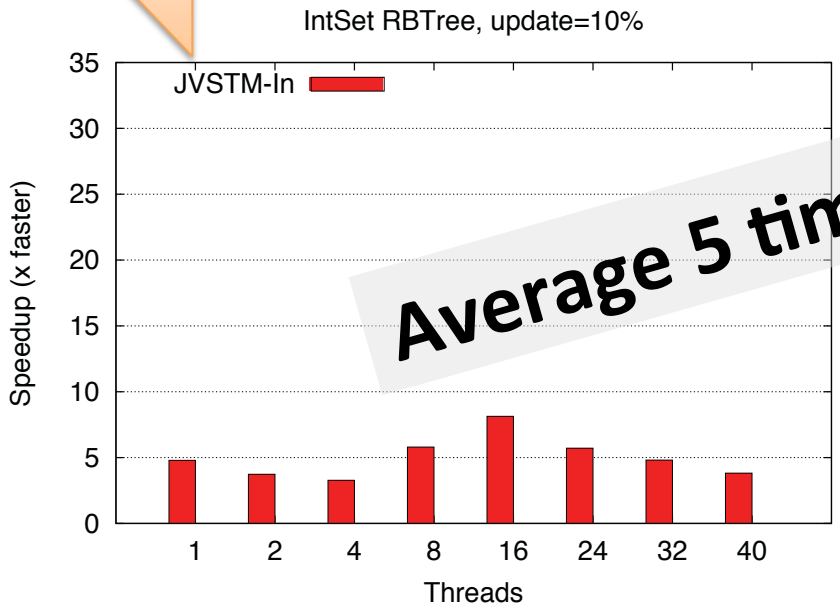
- Two implementations of JVSTM in DeuceSTM
  - Out-place strategy
    - Versions kept in external table with collision list
  - In-place strategy
    - No external table
    - Versions kept in meta-info field
- How much faster is the in-place implementation?

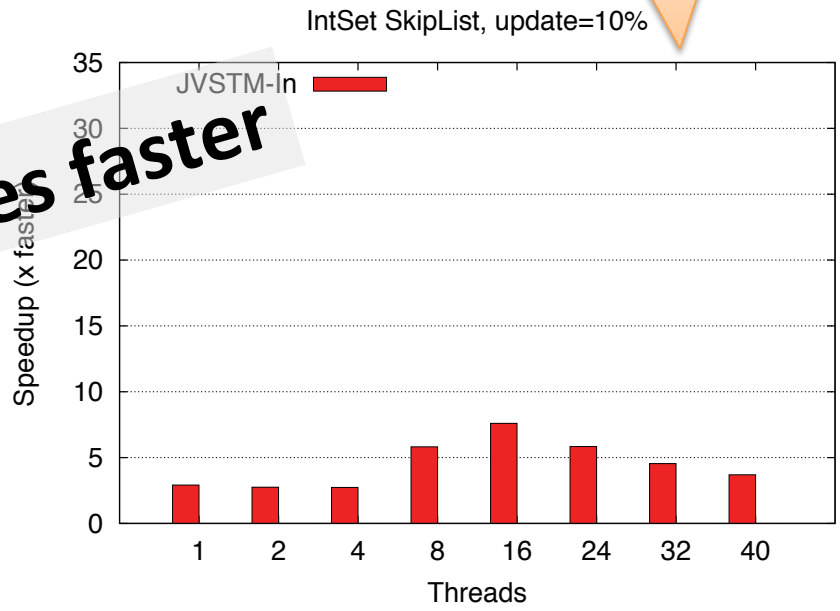
Experimental Evaluation
Multi-version

Not using Arrays

Write-Update: 10%

Using Arrays





Average 5 times faster

Speedup in X faster

Experimental Results
Multi-version

- JVSTM algorithm has a performance bottleneck in garbage collection of unused versions
  - Is it limiting the speedup for in-place?

- JVSTM-noGC: an extension of JVSTM where
  - Version lists are fixed sized
  - No garbage collection of unused versions

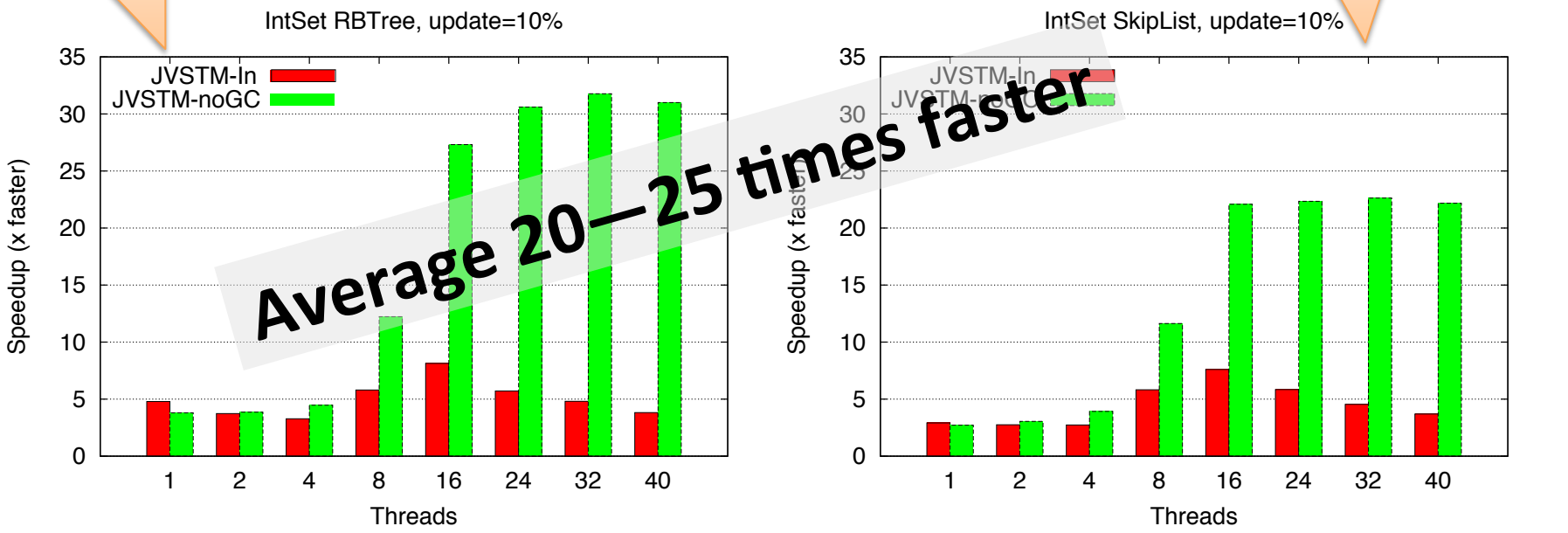
Experimental Evaluation

Multi-version

Not using Arrays

Write-Update: 10%

Using Arrays



Speedup in X faster

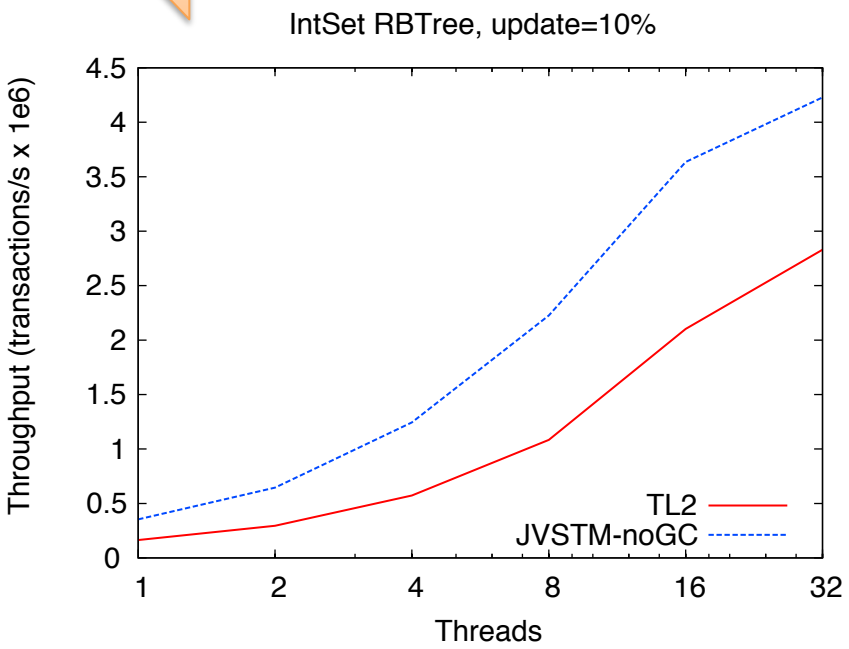
Experimental Evaluation

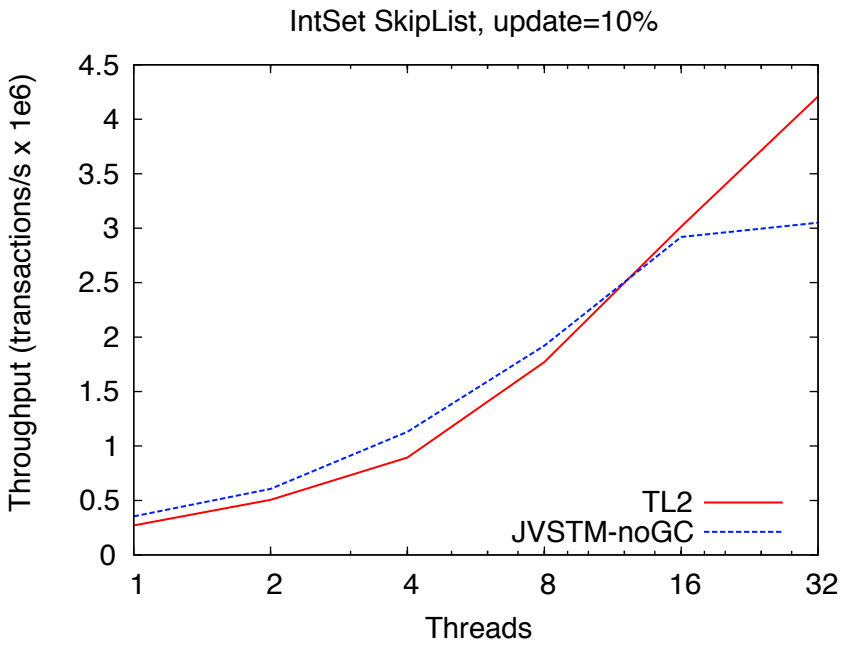
JVSTM vs TL2

Not using Arrays

Write-Update: 10%

Using Arrays





Concluding Remarks

- In-place strategy support allows:
  - Efficient support of primitive types
    - Avoids boxing and unboxing
  - Efficient Implementation of multi-version algorithms
  - Fair comparison of different kinds of STM algorithms
  - Support for distributed STM algorithms
    - Ongoing work

The End