Write-Write Conflict Detection for Distributed TM Systems

Ricardo J. Dias, Tiago M. Vale and João M. Lourenço
CITI / Universidade Nova de Lisboa

WDTM 2012, Lisbon, February 22, 2012

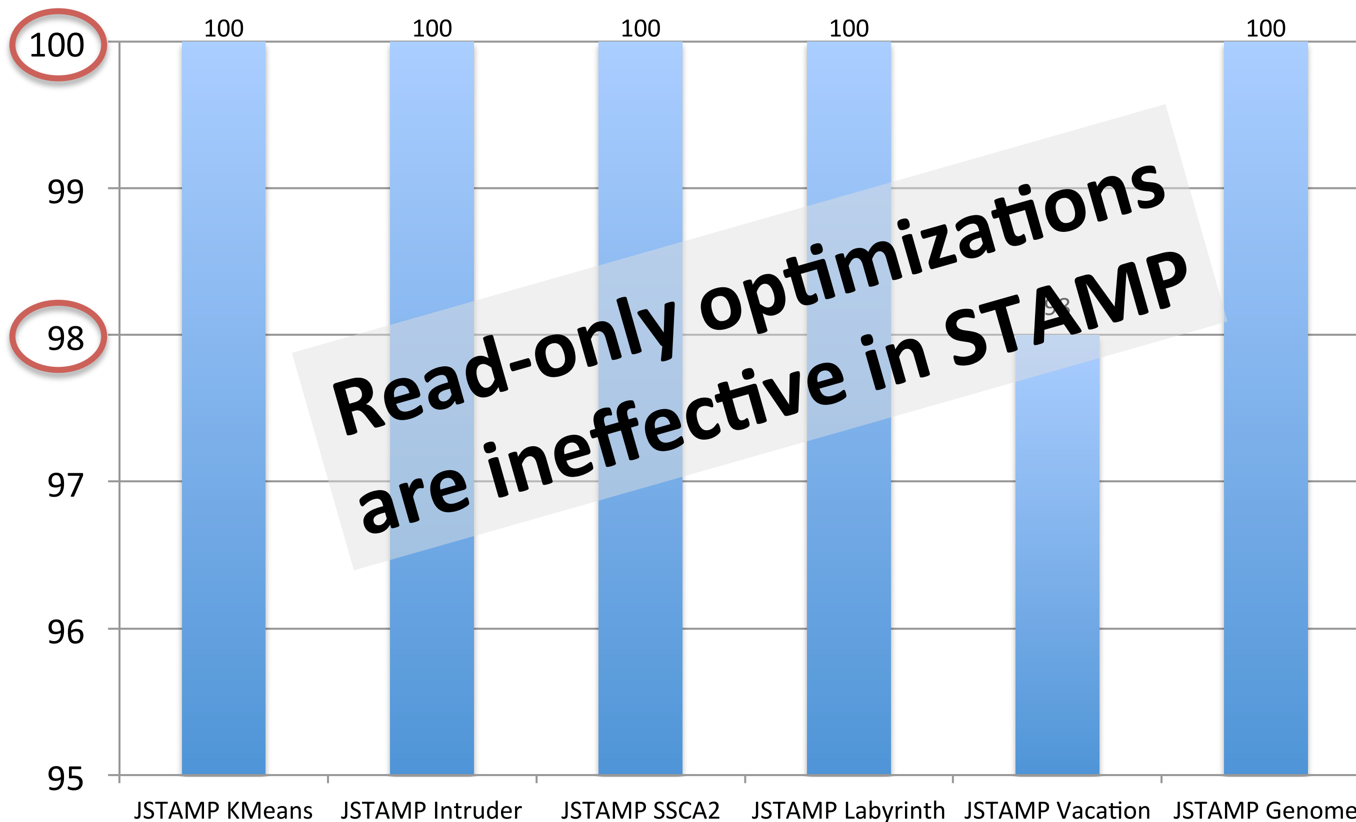
Motivation

- Applications have memory transactions with
  - Large read-sets
  - Small write-sets
  - Many read-only transactions
- True for some applications
  - Can we generalize?

Lets analyze some (STMAP) benchmarks!

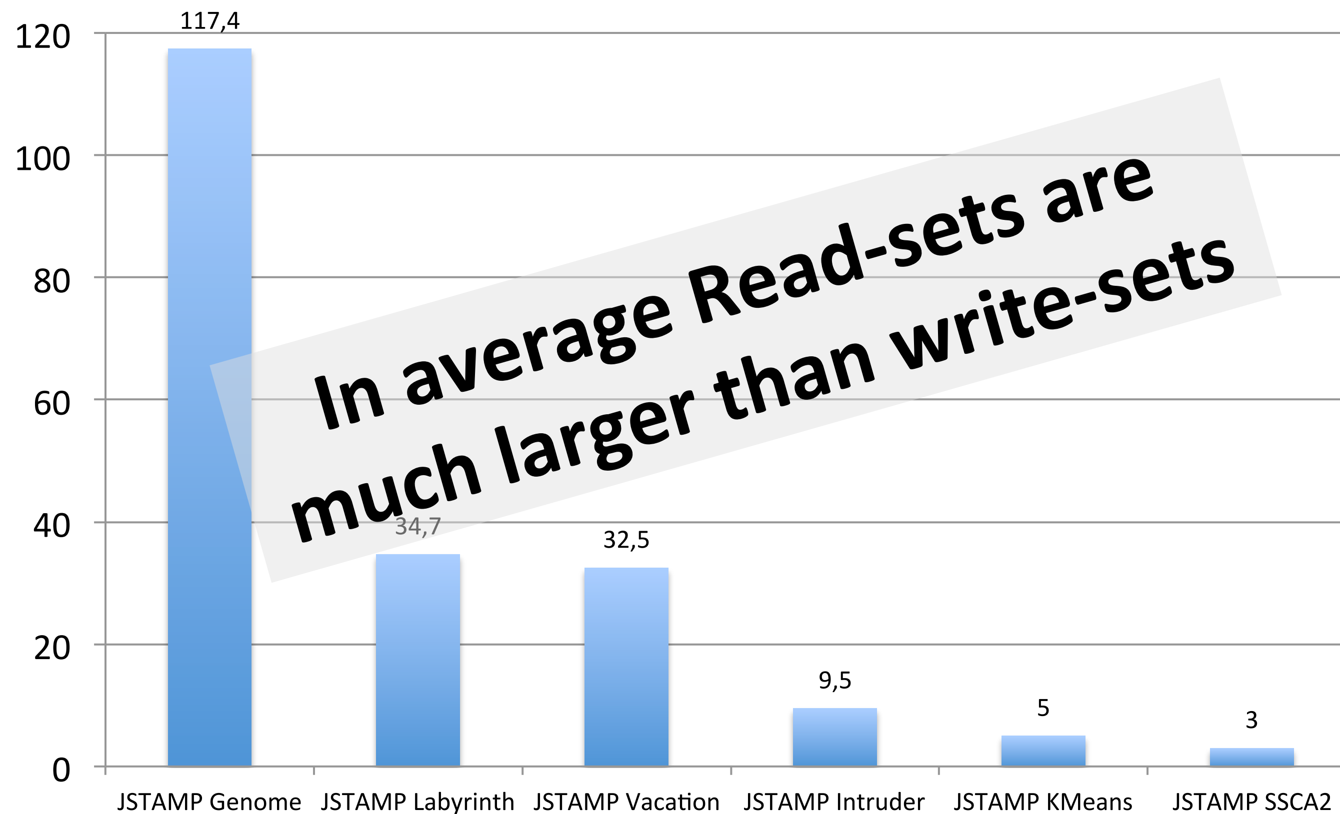
Statistics for STAMP

- Percentage of read-write transactions (over read-only)



Statistics for STAMP

- Average relative size of read-sets vs. write-sets (times larger)



Serialization Requires

- Interception and bookkeeping of:
  - Read and write operations
  - Bookkeeping cost: 2–8 times slower
- Validation of (small) write-sets against (large) read-sets
- DSTM: Remote validation
  - Increased network traffic due to broadcasting of read-sets

Advantages of Using SI

- No bookkeeping of read operations
- Faster validation at commit
  - Validation of (small) write-sets against write-sets, instead of (large) write-sets against read-sets

- DSTM: Remote validation
  - Avoid broadcasting (large) read-sets

Consistency Model

- Programmer (always?) expect serialization
- SI allows serialization anomalies
  - Write-Skew
- Detection of SI anomalies in possible at run-time
  - Incurs in bookkeeping and costs

Our Approach / Proposal

- Provide serialization semantics in the programming model
- Run-time supports a mixed-consistency model
  - Executes *certified* transactions in SI
  - Executes *non-certified* transactions in serialization
- Run-time base in vanilla JVMs

Our Approach / Proposal

- Automatic SI "anomaly free" certification
  - Use static analysis
- Very low impact in user's code
  - Addition of some (Java) annotation
- Can be complemented with dynamic/run-time detection of write-skew anomaly

Static Analysis

- Extraction of *abstract read-* and *write-sets* for each atomic block
- Abstract read- and write-sets
  - Sets of *abstract memory locations*
  - Over- and under-approximations needed
- Abstract memory locations (*heap paths*)
  - Restricted Regular Expressions
    - { head.next*.value }
    - { root.(left | right)*.right }

Static Analysis — Soundness

- Runtime Write-Skew:

$$\mathcal{R}_1^c \cap \mathcal{W}_2^c \neq \emptyset \quad \wedge \quad \mathcal{W}_1^c \cap \mathcal{R}_2^c \neq \emptyset \quad \wedge \quad \mathcal{W}_1^c \cap \mathcal{W}_2^c = \emptyset$$

- Assumptions:

$$\mathcal{R}_1^c \subseteq \mathcal{R}_1, \quad \mathcal{R}_2^c \subseteq \mathcal{R}_2, \quad \mathcal{W}_1^c \subseteq \mathcal{W}_1^>, \quad \mathcal{W}_2^c \subseteq \mathcal{W}_2^>, \quad \mathcal{W}_1^< \subseteq \mathcal{W}_1^c, \quad \mathcal{W}_2^< \subseteq \mathcal{W}_2^c$$

- Soundness

$$(\mathcal{R}_1^c \cap \mathcal{W}_2^c \neq \emptyset \quad \wedge \quad \mathcal{W}_1^c \cap \mathcal{R}_2^c \neq \emptyset \quad \wedge \quad \mathcal{W}_1^c \cap \mathcal{W}_2^c = \emptyset) \Rightarrow$$
$$(\mathcal{R}_1 \cap \mathcal{W}_2^> \neq \emptyset \quad \wedge \quad \mathcal{W}_1^> \cap \mathcal{R}_2 \neq \emptyset \quad \wedge \quad \mathcal{W}_1^< \cap \mathcal{W}_2^< = \emptyset)$$

Static Analysis - Usage

- Abstract Read- and Write-Sets can be used for:
  - Detecting write-skew anomalies
  - Detecting write-write conflicts

  - Can also be used in other domains

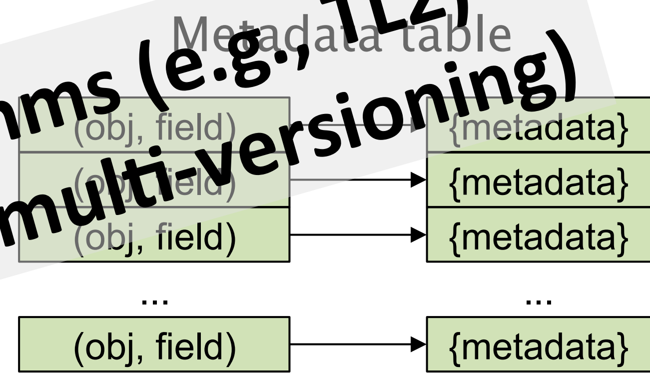
Static Analysis — Prototype

- Java bytecode analysis
- Requires the specification of the initial heap state for each atomic block
  - Separation Logic specification
- The analyses supports
  - Tree-based acyclic data-structures

Run-Time Support — DeuceSTM

- Support for in-place metadata
  - Full support for primitive types
  - Efficient solution for arrays
- Appropriate for supporting multi-version in DeuceSTM
- Metadata fields will support DSTM specific metadata (e.g., global object ID)

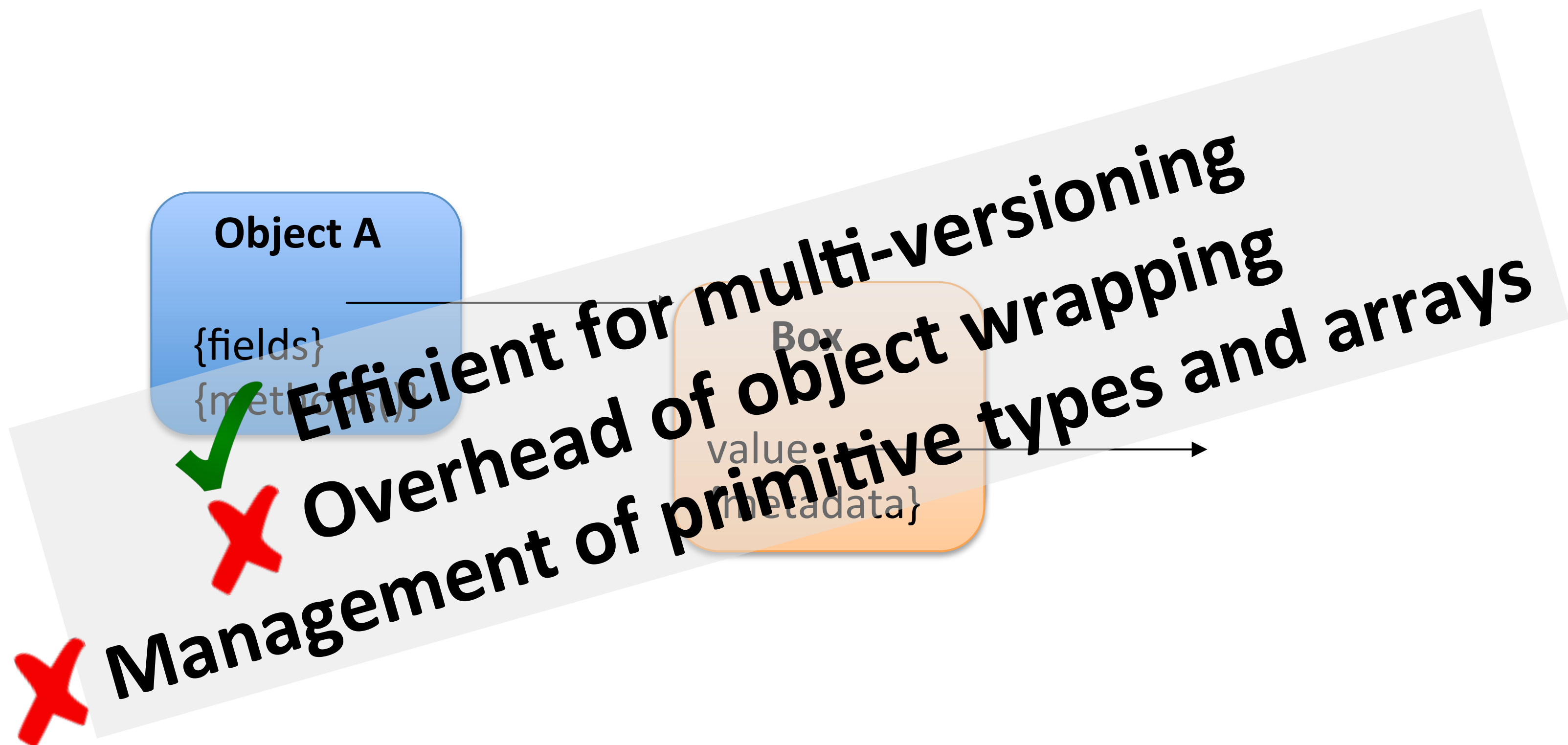
Out-Place Metadata



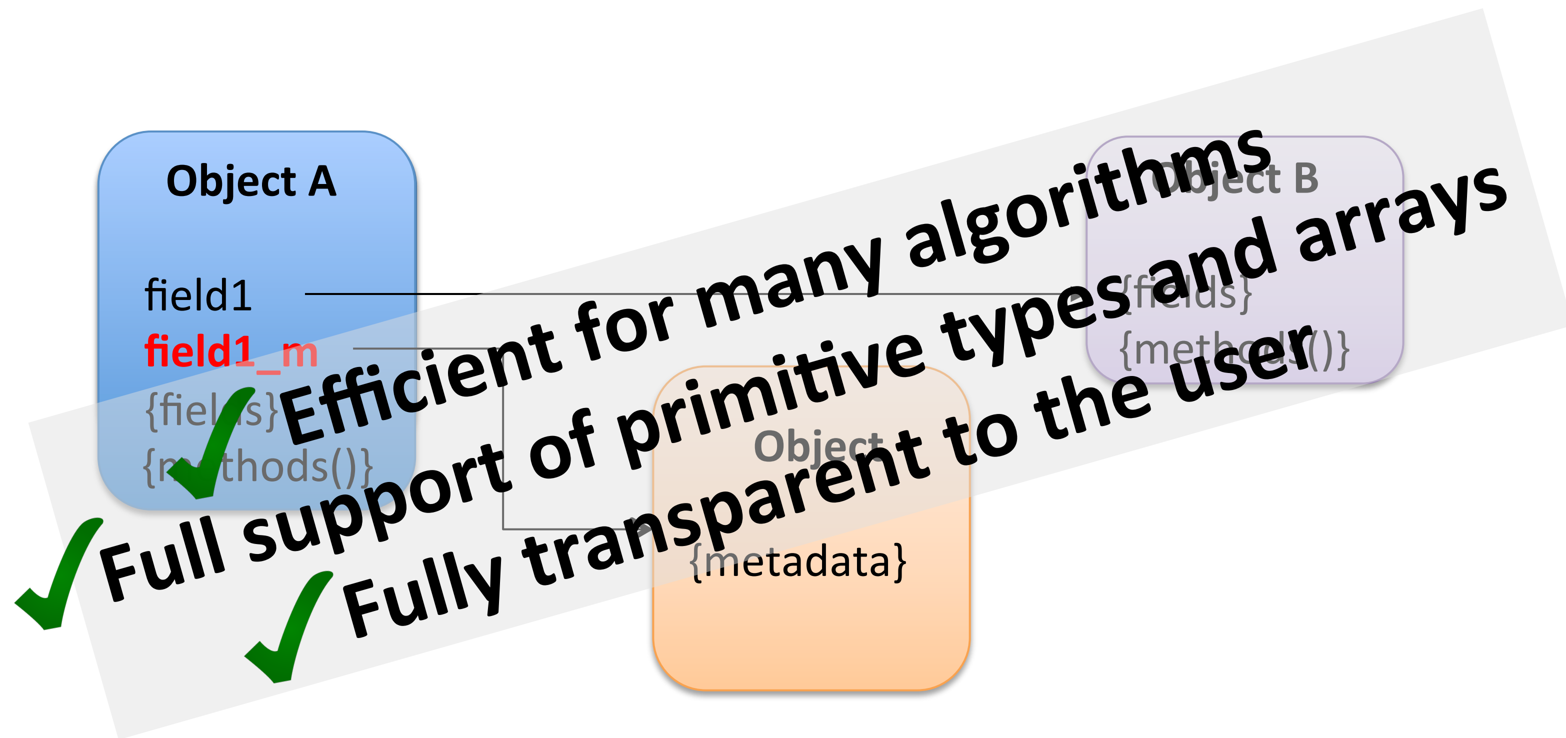
✓ Efficient for some algorithms (e.g., TL2)

✗ Inefficient for others (e.g., multi-versioning)

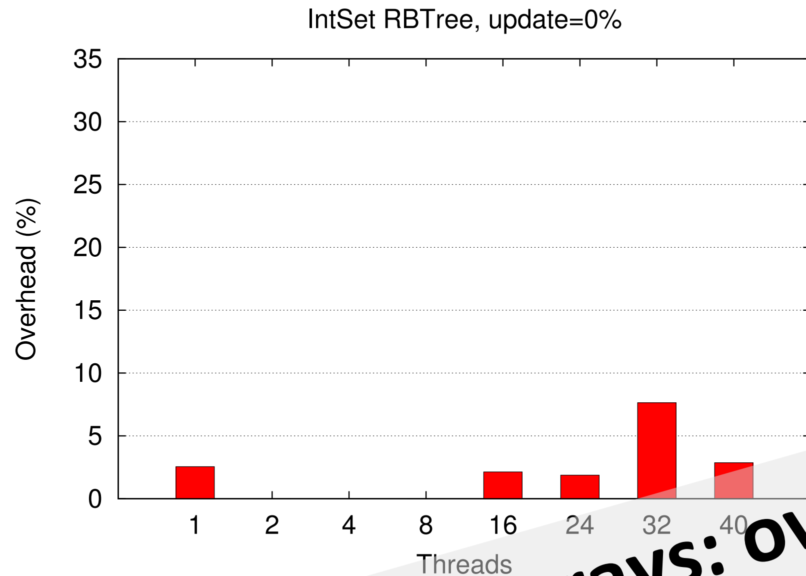
In-Place Metadata—Classical Approach

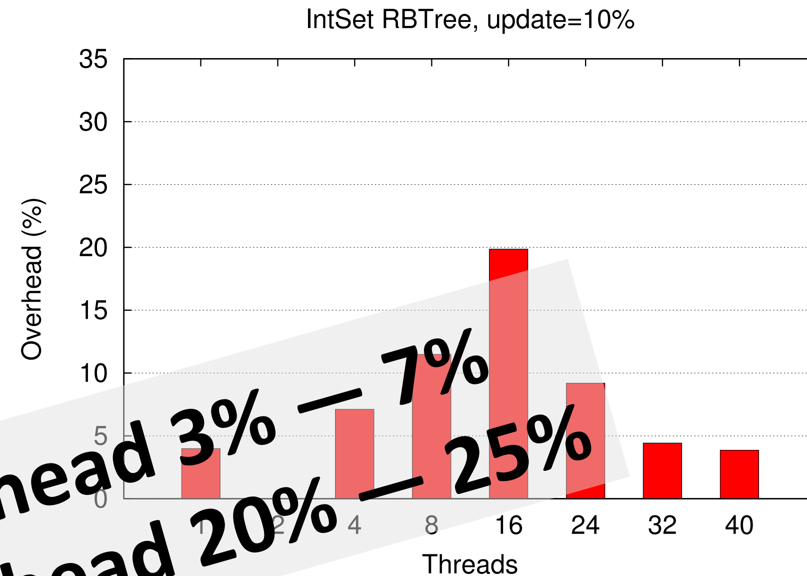


In-Place Metadata—Our Approach



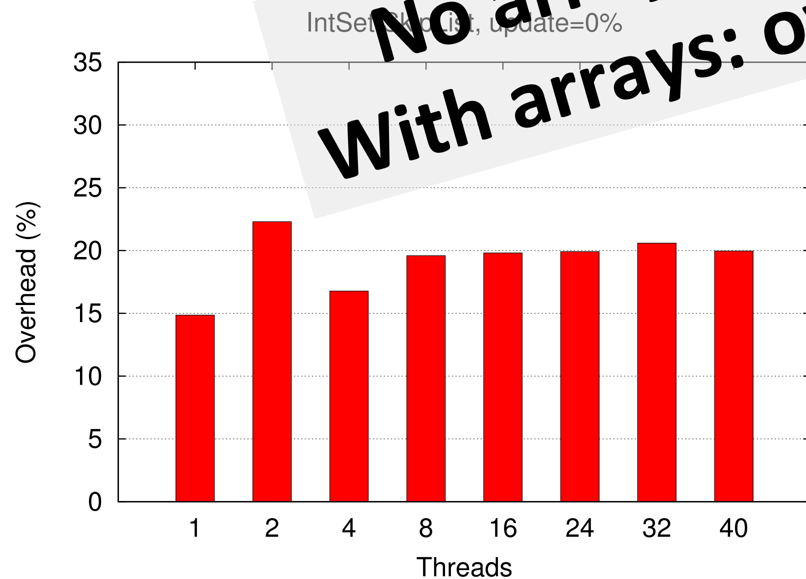
Overhead: DeuceSTM + In-place Metadata

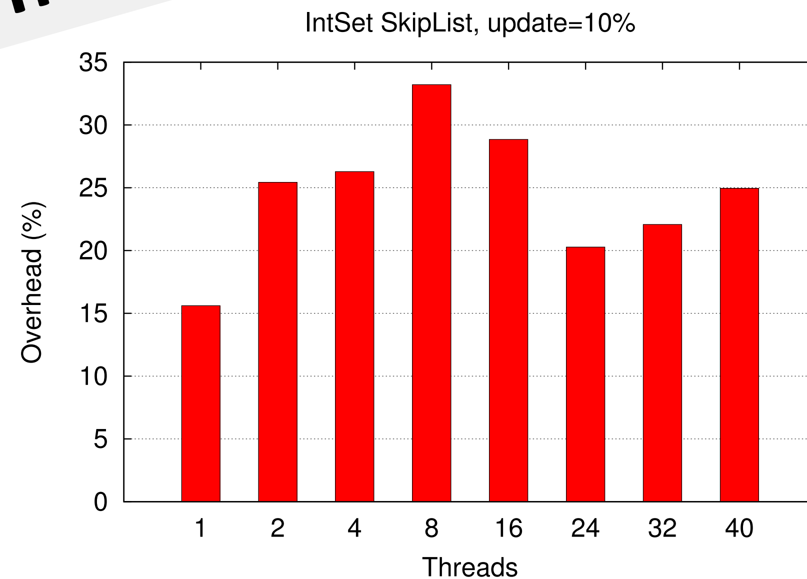




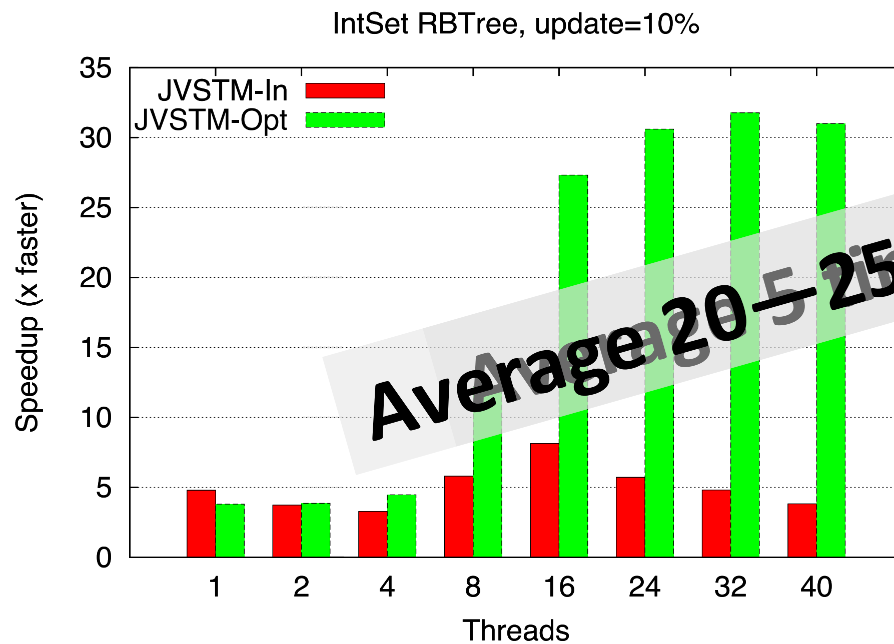
No arrays: overhead 3% — 7%

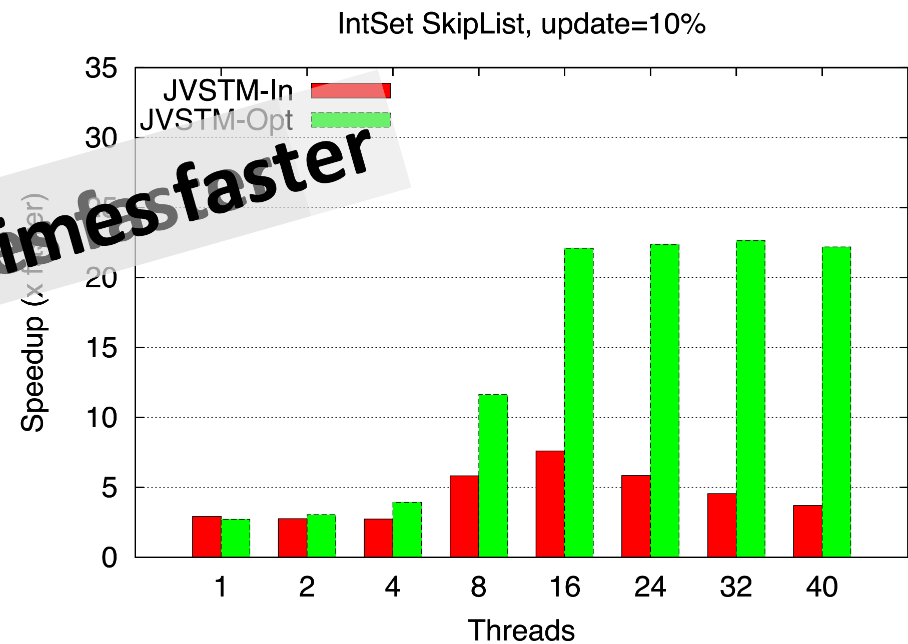
With arrays: overhead 20% — 25%





Performance: DeuceSTM + + In-place Metadata + Multiversioning





Average 20–25 times faster

Current Status

- Static analysis framework for detection of write-skew anomalies
  - Sound analysis
  - Java bytecode
  - Tree-based acyclic data-structures

Current Status

- Run-time framework
  - Extension of DeuceSTM for in-place metadata
  - Efficient support for primitive types and arrays
  - Efficient support for multi-versioning
  - Serialization semantics
  - No significant changes in user's code

Summary

- Aim at providing serialization semantics with mixed-consistency run-time
  - Efficient support for mixed consistency mode
- On-going work towards a generic DSTM framework
  - Solid work on the STM infrastructure
  - Ongoing work on the DSTM support
- System in not fully-assembled yet