On the Platform Specificity of STM Instrumentation Mechanisms

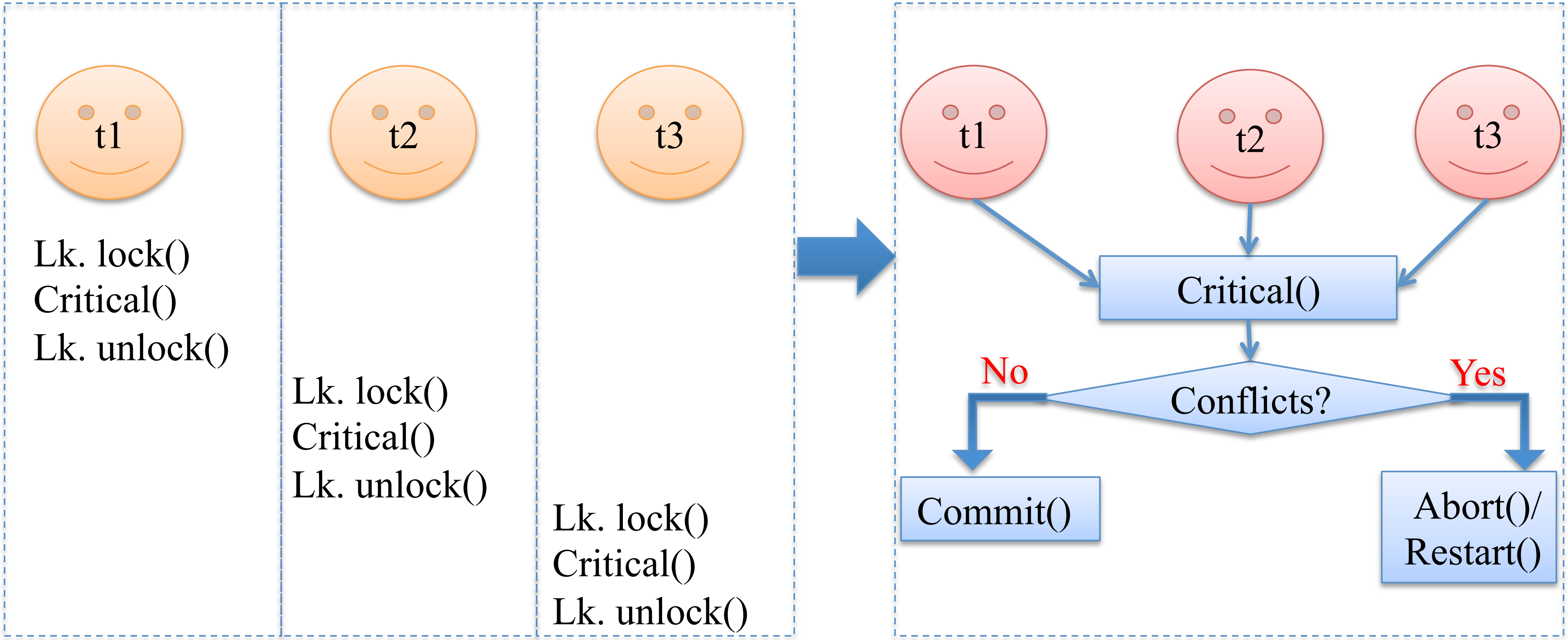
Wenjia Ruan, Yujie Liu, Chao Wang, and Michael Spear

Computer Science and Engineering Department

Lehigh University

Background: Transactional Memory (TM)

- Mechanism to simplify concurrent programming
- Replace critical sections with transactions
- Transactions use speculation under the hood



t1

Lk. lock()
Critical()
Lk. unlock()

t2

Lk. lock()
Critical()
Lk. unlock()

t3

Lk. lock()
Critical()
Lk. unlock()

t1 t2 t3

Critical()

Conflicts?

No

Yes

Commit()

Abort()/
Restart()

Implementing Software TM (STM)

Programmer writes code:

```
__transaction_atomic {
  if (B == 0)
      A ← 1
  else
      B ← 0
}
```

Compiler generates code:

```
TM_CHECKPOINT()
TM_BEGIN()
if (TM_READ(B) == 0)
  TM_WRITE(&A, 1)
else
  TM_WRITE(&B, 0)
TM_COMMIT()// may fail
           // and cause
           // txn to
           // retry
```

- Many possible implementations of the TM_ functions

TM Availability and Use

- Latest product announcements from Intel and IBM include **Hardware TM** (HTM) support
  - Not all transactions can be executed using HTM (e.g., big transactions, long-running transactions)
  - HTM can enter pathological situations (e.g., livelock)
  - Software fallback is required
  - Lots of legacy systems would still need STM if they want to use transactional memory
- **Software TM** progress
  - C++ standardization effort underway since 2009
  - Support in Scala, Clojure, Haskell

This Paper

- How does the **platform** (OS, CPU) affect the manner in which a compiler should instrument code to achieve efficient STM?
  - Focus on platform characteristics and relationship to system Application Binary Interface (ABI)
  - Goal: to see what implementation details have platform-specific overheads
- Evaluations on 4 platforms:
  - ARM/Linux
  - SPARC/Solaris
  - IA32/Linux
  - IA32/MacOS

Main Results

- Many sources of overhead are platform-specific:
  - Accessing thread-local storage (TLS)
  - Mechanisms for adaptivity/autotuning
  - Memory consistency model / memory fences

- Achieving optimality on every platform is a code maintenance nightmare
  - Macros to generate function signatures
  - Extend compiler toolchain to produce function bodies for adaptive libraries
  - Best algorithm for one platform can be worst on another platform
    - We designed a new STM algorithm “Cohorts”

Generality

- These issues are not specific to STM
  - NUMA-aware algorithms often require TLS access
  - Autotuning and adaptive libraries are common in HPC
  - The world isn’t all x86/Linux

Evaluating TLS Overheads

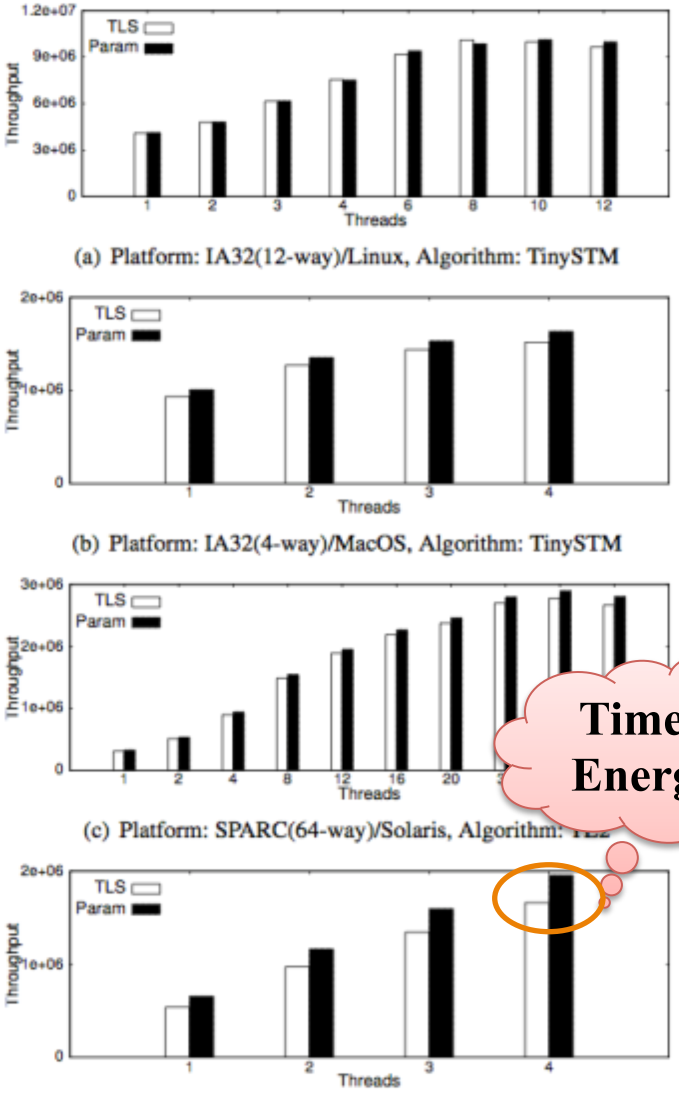
- STM implementations use **per-thread metadata** (e.g., “transaction descriptors”) to store read/write sets, rollback values, etc.
  - Accessed on every shared memory load/store

- Common ways to access per-thread metadata:
  - Access thread-local storage (**TLS**). (E.g., use __thread modifier) --- used currently by GCC compiler.
  - Compiler explicitly manage descriptors. (e.g., the descriptor is passed to the TM library as an **additional parameter**) --- used currently by Oracle TM compiler.
    - Requires changes to function signatures

Evaluating TLS Overheads

- We modified RSTM to support both conventions
- Executed stress-tests, where there is no work between transactions
  - Red/black tree, equal mix of insert and remove operations, 8-bit values, 50% update transactions
  - Use gcc in all cases

Evaluating TLS Overheads



(a) Platform: IA32(12-way)/Linux, Algorithm: TinySTM

(b) Platform: IA32(4-way)/MacOS, Algorithm: TinySTM

(c) Platform: SPARC(64-way)/Solaris, Algorithm: TL2

(d) Platform: ARM(4-way)/Linux, Algorithm: NOrec

Time is Energy!

IA32/ Linux

No noticable difference

```
get_tls:
mov  %gs:0xfffffff0, %eax
```

IA32/ MacOS

5% slowdown

No OS-level support for __thread…
Must use pthread_getspecific()

SPARC/ Solaris

3% slowdown

```
get_tls:
sethi  %hi(0),  %g1
xor    %g1, -4, %g1
ld    [ %g7  + %g1 ], % o0
```

ARM/ Linux

15% slowdown!

```
get_tls:
ldr        r3,    [pc, #12]
push   { lr }
bl          __aeabi_read_tp
. . .
```

7 instructions, Including a branch

Summary of TLS costs

- **Performance**:
  - On some platforms, __thread has noticeably more latency, on others it is faster
- **Software engineering / code maintenance**:
  - Optimal solution requires different interfaces to a library depending on the architecture
  - Use #defines for every function in a library

Accessing the TM Library

There is no single TM implementation suitable for all circumstances.

- At least limited **adaptivity** is advisable
  - C++ spec needs it
  - Changing mode/algorithm for performance
- Even more likely when HTM arrives

Accessing the TM Library

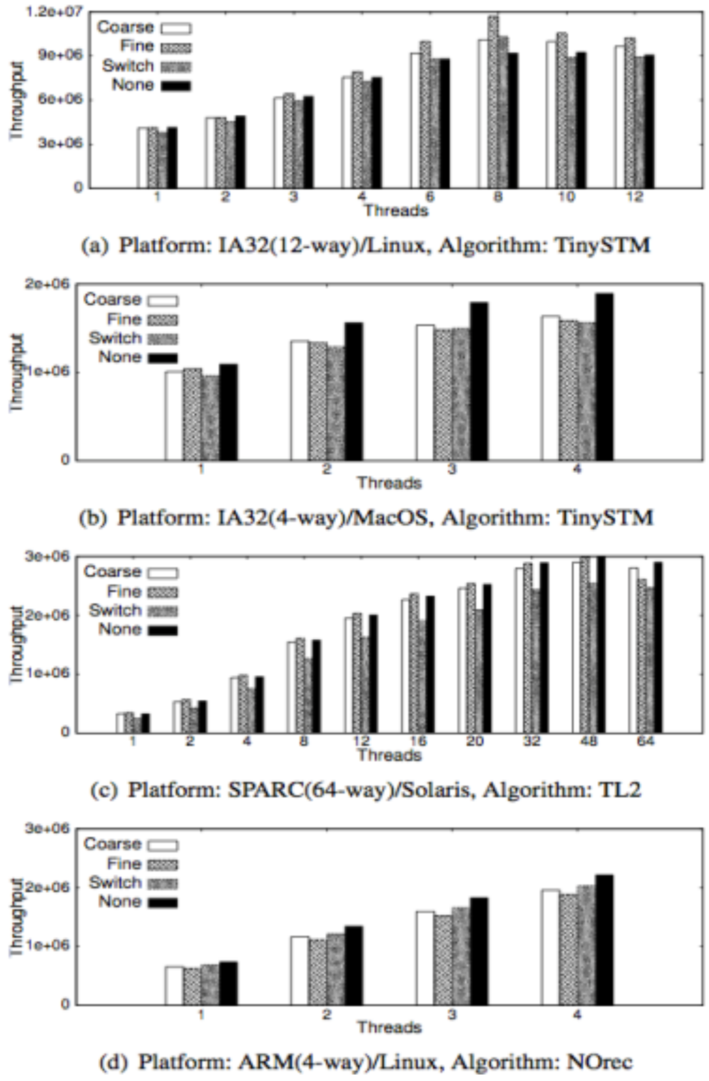
Three common approaches for supporting adaptivity within a TM implementation:

1. **Coarse**: use global function pointers, which coordinate all transactions' behavior without incurring additional branching overhead in the common case

2. **Fine**: use per-thread function pointers, so each thread can make fine-grained decisions based on previous access patterns

3. **Switch**: use conditionals (typically a "switch" statement) to globally coordinate which function to call

Accessing the TM Library



(a) Platform: IA32(12-way)/Linux, Algorithm: TinySTM

(b) Platform: IA32(4-way)/MacOS, Algorithm: TinySTM

(c) Platform: SPARC(64-way)/Solaris, Algorithm: TL2

(d) Platform: ARM(4-way)/Linux, Algorithm: NOrec

Coarse

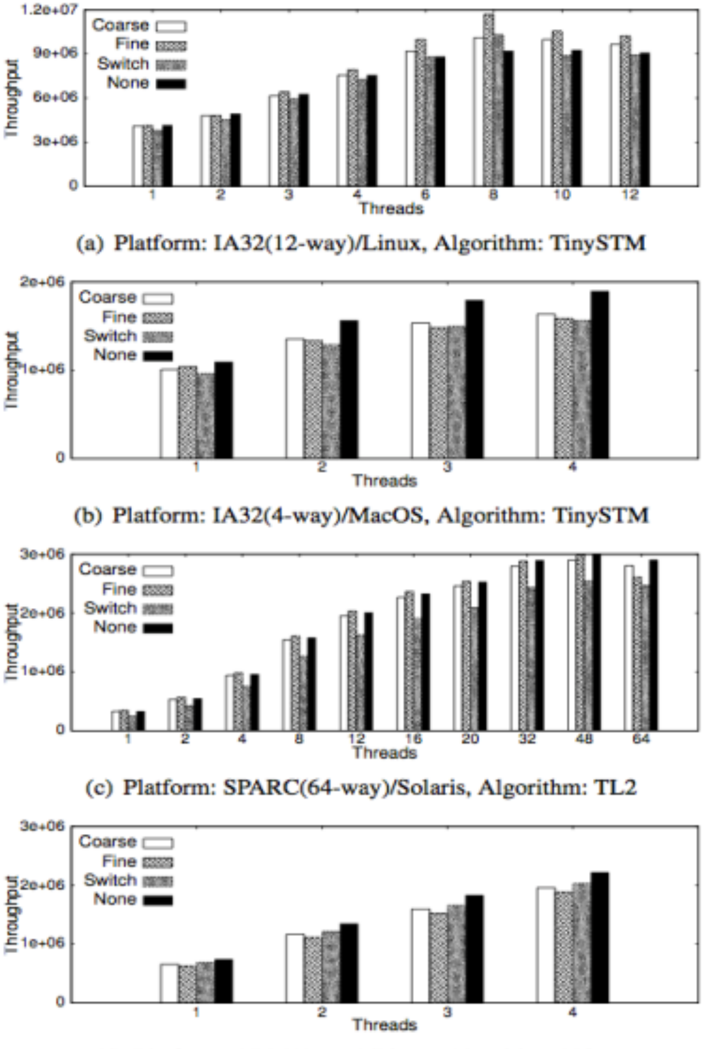
- **Global function pointers**
- **Avoid TLS overhead, prevents branching in the common case where adaptivity is not occurring.**

```
tmread = LSA.read;
tmwrite = LSA.write;
tmcommit = LSA.commit;
```

Optimal on IA32/MacOS:

- TLS is expensive.
- Indirection of function pointers does not incur a noticeable cost

Accessing the TM Library



(a) Platform: IA32(12-way)/Linux, Algorithm: TinySTM

(b) Platform: IA32(4-way)/MacOS, Algorithm: TinySTM

(c) Platform: SPARC(64-way)/Solaris, Algorithm: TL2

(d) Platform: ARM(4-way)/Linux, Algorithm: NOrec

Fine

- **Per-thread function pointers**
- **Affords maximum flexibility**
- **Incurs TLS overhead to locate function pointers**

```
*(threads[i]->my_tmread) = (void*)LSA.read_ro;
*(threads[i]->my_tmwrite) = (void*)LSA.write_ro;
*(threads[i]->my_tmcommit) = (void*)LSA.commit_ro;
```

Best for IA32/Linux and SPARC/Solaris:

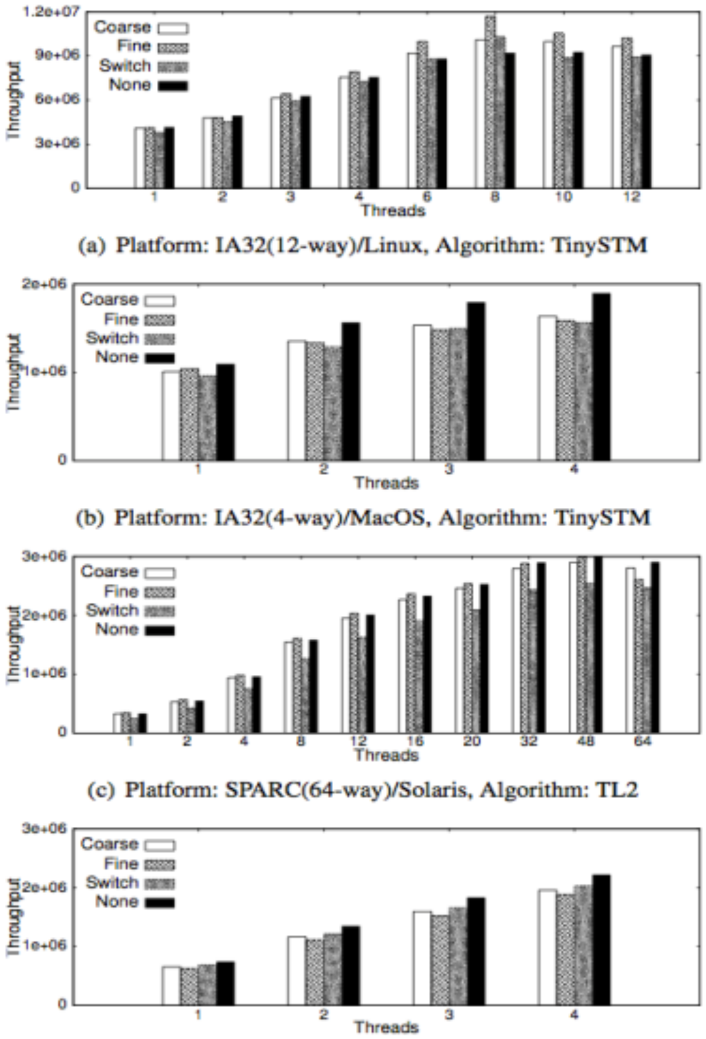
[IA32]

- Indirect calls and TLS are inexpensive.
- Even better than “None” (avoids branches to recover more performance than it costs).

[SPARC]

- Greatly benefits from the reduction in branches

Accessing the TM Library



(a) Platform: IA32(12-way)/Linux, Algorithm: TinySTM

(b) Platform: IA32(4-way)/MacOS, Algorithm: TinySTM

(c) Platform: SPARC(64-way)/Solaris, Algorithm: TL2

(d) Platform: ARM(4-way)/Linux, Algorithm: NOrec

Switch

- **Use a “switch” statement to choose the right instrumentation.**
- **No TLS or indirect call overhead**
- **Should be fast on architectures with inexpensive indirect branches**

```
void tmwrite(addr, val) {
 switch (alg) {
  case LSA:  LSAWrite(addr, val); break;
   …
}
```

Optimal on ARM/Linux:

- TLS is expensive.
- Jump tables have low latency

Summary of Library Access Costs

- Nuanced interaction between TLS and mode-switching
  - More factors than just the cost of TLS
- Not specific to TM library, likely to affect any adaptive or auto tuned library

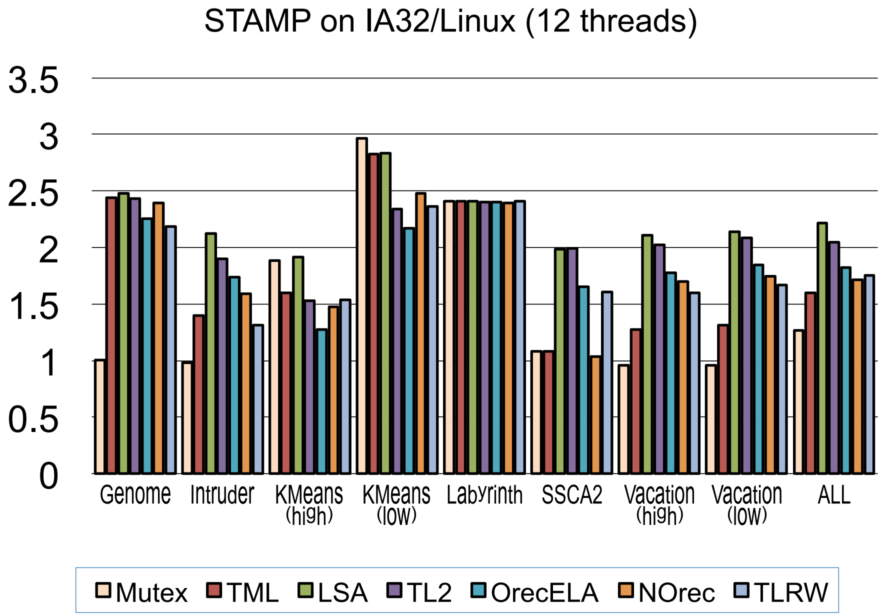
Algorithm Selection and Optimization

- Architecture influences which algorithms are worth considering in an auto tuned STM

E.g.:

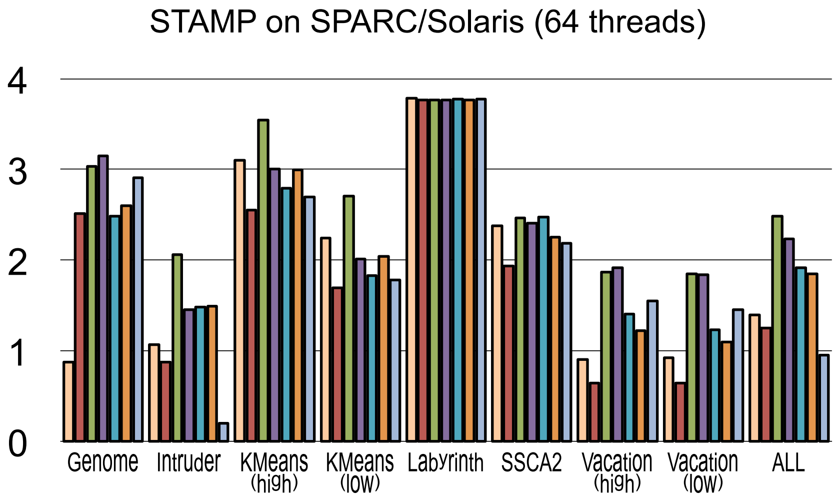
- Number of cores
- Cost of CAS instructions
- Cost of Memory fences on each read/write
  - Usually ignored with x86 and SPARC
  - but significant on ARM and POWER

Evaluation on IA32/Linux and SPARC/Solaris



1. **LSA**, **TML** and **TLRW** are the only algorithms in which locks are acquired before commit time.

2. **TML**, **OrecELA** and **NOrec** are the only algorithms that comply with the C++ memory model.



- **LSA** offers best performance
- **OrecELA** offers best performance with C++ compatibility

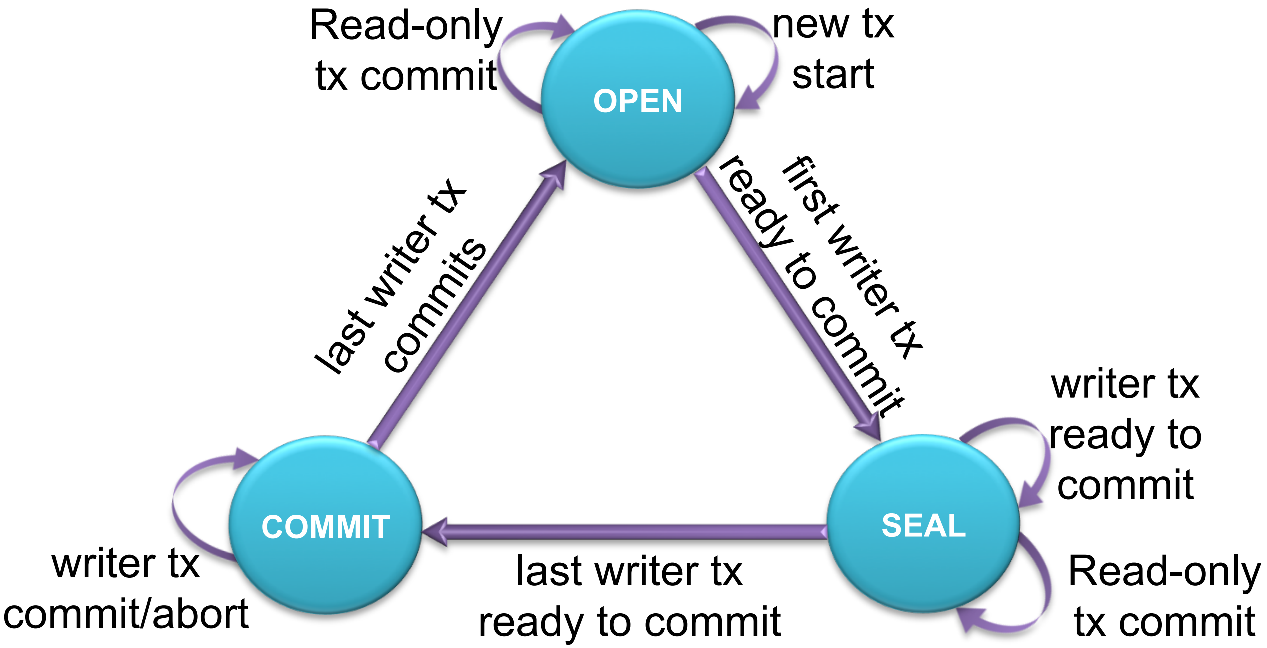
These are already known. What about on ARM?

Algorithm Selection on ARM

- **Does ARM represent a unique design point?**
  - LSA and TL2: each load requires two fences.
  - NOrec, TML and OrecELA: one fence is required after each load.
  - TLRW requires one fence on every load and store.
  - Fences are expensive!

- Our new algorithm Cohorts requires **zero** fences load/store
  - Only good for low-core counts
  - Only beneficial due to eliminating fences
  - Performs badly on x86 and SPARC

Cohorts: A New STM Algorithm for ARM



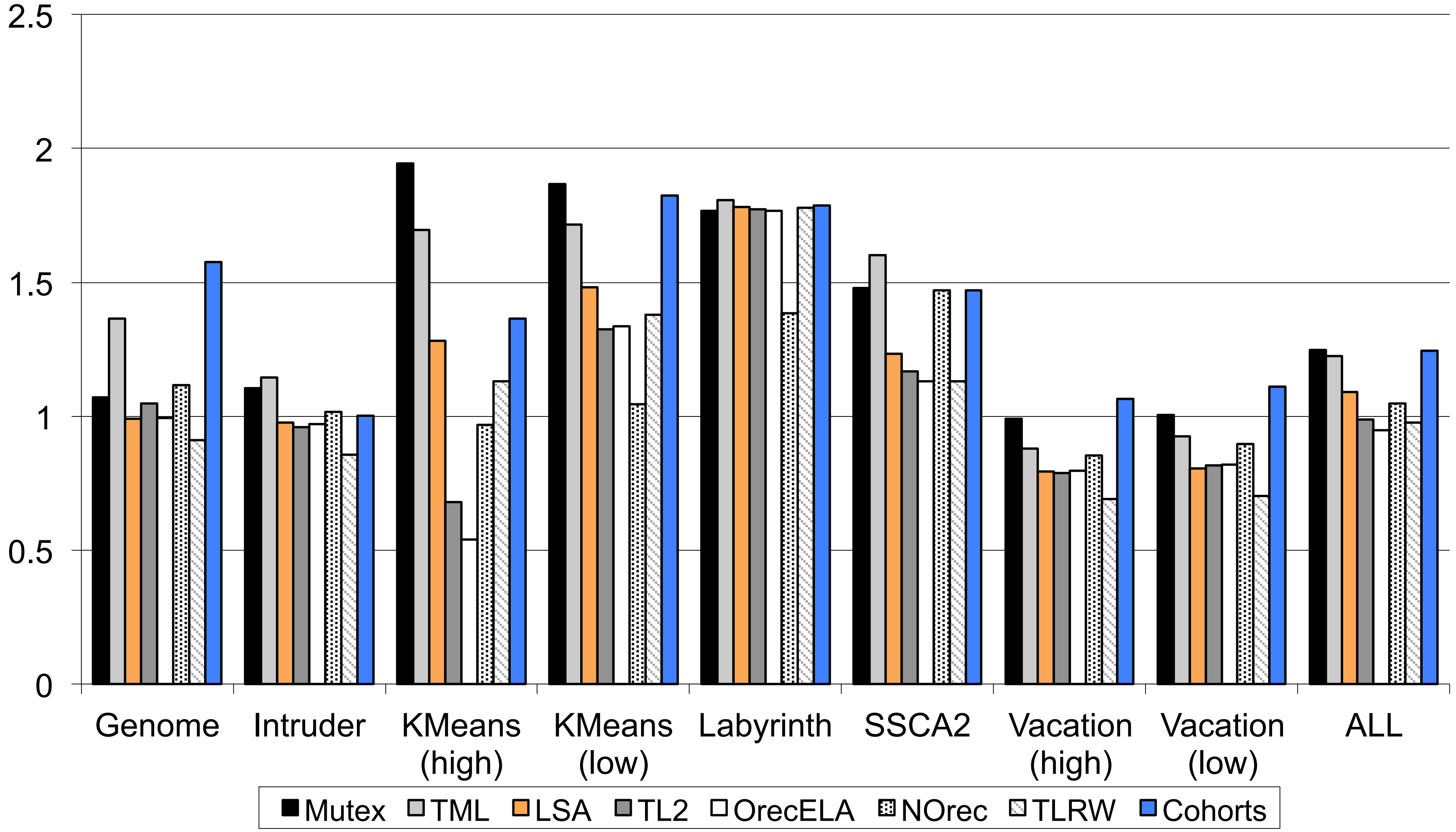
- Three phases, transactions progress as a cohort
  - Once one writer ready to commit, none can start
  - Once all writers ready to commit, commit one at a time
  - Once all writers finish commit, everyone can start again
- Memory is immutable during execution, no fences needed

Cohorts: Simplified TM_READ()

```
TL2_TM_READ(addr) {
  // check if exist in write set
  if (addr $\in$ writes)
     return writes[addr];

  // check metadata
  o1 $\leftarrow$ orecs[addr].getValue();
  FENCE;
  // get value of the address
  v $\leftarrow$ *addr;
  FENCE;
  // check metadata
  o2 $\leftarrow$ orecs[addr].getValue();

  // validate
  if (o1 = o2 and o2 $\leq$ start) {
    reads $\leftarrow$ reads $\cup$ {addr};
    return v; }
  else ABORT();
}
```

```
Cohorts_TM_READ (addr)  {
  // check if exist in write set
  if (addr $\in$ writes)
    return writes[addr];

  // get value of the address
  v $\leftarrow$ *addr;
  reads $\leftarrow$ reads $\cup$ {$\langle$addr, v$\rangle$};
  return v;
}
```

Evaluation on ARM/Linux (Tegra3: 4 threads)



- Cohorts gives best overall performance
- Still not significantly better than Mutex

Cohorts: Future Work

- Many cohort-specific compiler optimizations are possible
- Possibly a good fit for Xbox?
- But HTM for ARM would be a better approach

Conclusions

- Paper focuses on engineering issues
  - Demonstrates that run-time costs of library interface are platform-specific
  - Implications for any multi-platform concurrent or autotuned library
  - Software engineering nightmare to try to provide best alternative in every case
- Some easy fixes (MacOS needs better TLS), but not others
  - Cohorts: a first attempt for STM algorithm on ARM
  - Worst case: need to rethink concurrent algorithms in ARM version of the code

Thank you …

- Open sourced, code available at Google Code repository:
  - http://code.google.com/p/rstm/
- Contact Info:
  - **W**enj**i**a **R**uan: rwj@lehigh.edu
- Questions?