Using Shared Memory Abstractions to Design Eager Sequentializations for Weak Memory Models

Ermenegildo Tomasco[1], Truc L. Nguyen[1],
Bernd Fischer[2], Salvatore La Torre[3], Gennaro Parlato[1]

[1] University of Southampton, United Kingdom
[2] Stellenbosch University, South Africa
[3] Università degli Studi di Salerno, Italy

UNIVERSITEIT·STELLENBOSCH·UNIVERSITY
jou kennisvennoot • your knowledge partner

How to Build Bounded Model Checkers for Concurrent Programs on the Cheap?

Use source-to-source transformation!

Ermenegildo Tomasco[1], Truc L. Nguyen[1],
Bernd Fischer[2], Salvatore La Torre[3], Gennaro Parlato[1]

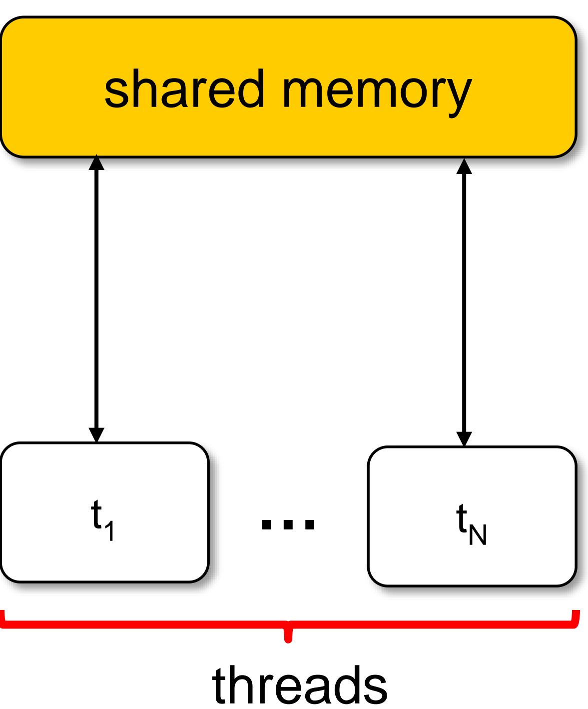
[1] University of Southampton, United Kingdom
[2] Stellenbosch University, South Africa
[3] Universita degli Studi di Salerno, Italy

UNIVERSITEIT•STELLENBOSCH•UNIVERSITY
jou kennisvennoot • your knowledge partner

Shared Memory Concurrency

Computational model:

- independent computation threads
- (unsychronized) read/write access to shared memory



Sequential consistency:

- changes to the shared memory immediately visible to all threads
- memory operations executed in program order within each thread
- relatively simple to reason about but not realistic

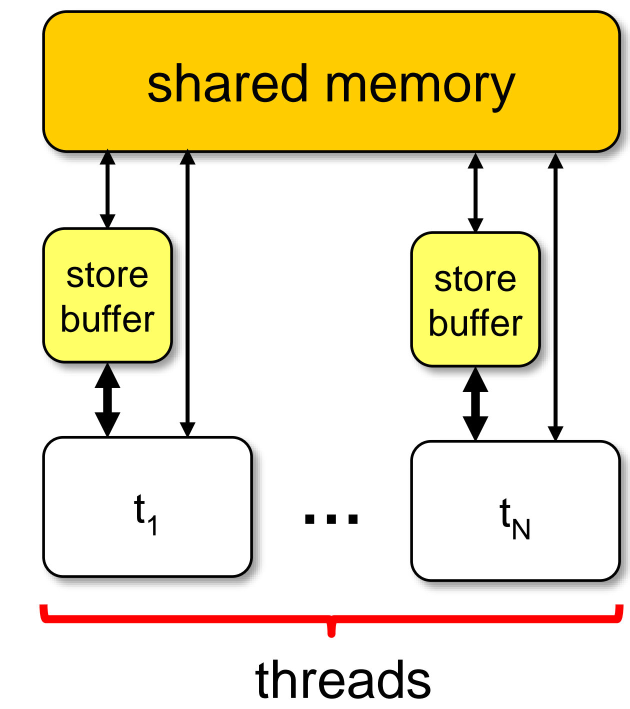
Weak memory (relaxed consistency) models:

- used in practice to exploit modern hardware
- memory operations may be reordered

Weak Memory Models (WMMs)

Computational model:

- fast store buffers for threads
- non-deterministic synchronization with shared memory



Total Store Order (TSO):

- one buffer per thread
- reads may overtake writes
- writes executed in order (per thread)

Partial Store Order (PSO):

- one buffer per variable per thread
- reads may overtake writes
- writes to different variables may be reordered
- writes to the same variable executed in order (per thread)

Bug-finding and Verification under WMMs

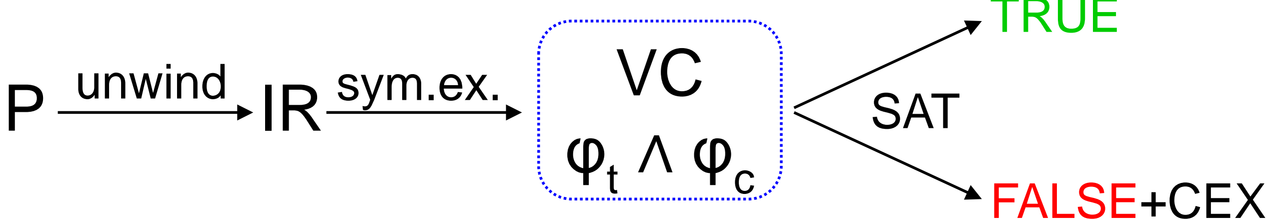
Testing:

- generally ineffective for rare concurrency errors
- cannot control (hardware-level) nondeterminism added by WMMs
- needs to be complemented with symbolic analysis

Bug-finding and Verification under WMMs

Bounded model checking (BMC):

- concurrency handling at formula level

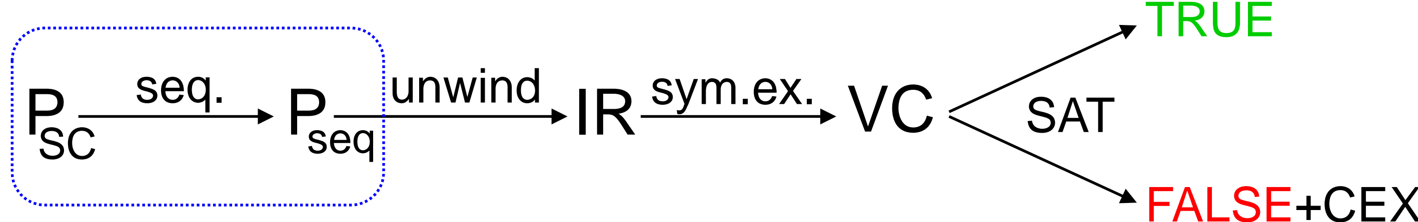


  - encode threads separately
  - add φ_c to capture thread interleaving [Sinha/Wang, POPL'11]
- extension to WMMs is natural…
  - change φ_c to capture extra interactions by WMM [Alglave et al., CAV'13]
- … but intrusive, with tool lock-in

Bug-finding and Verification under WMMs

Sequentialization (+ BMC):

- concurrency handling at source-code level

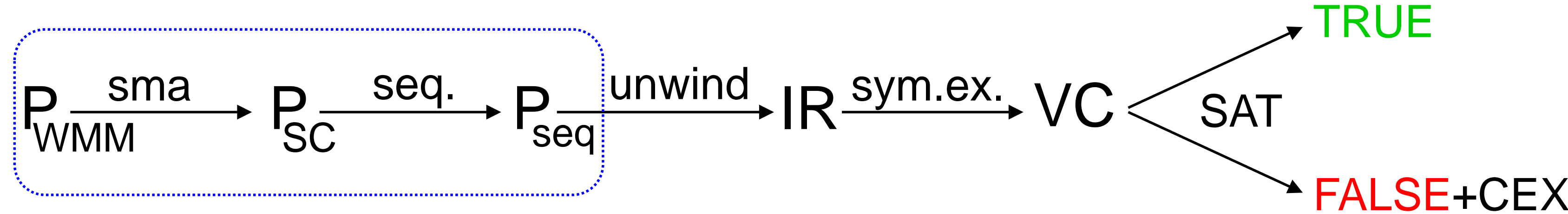


  - reduction to sequential programs analysis
  - implemented as non-intrusive source transformation
  - lazy sequentialization for effective bug-hunting [Inverso et al., CAV'14]

Bug-finding and Verification under WMMs

Sequentialization (+ BMC):

- concurrency handling at source-code level



  - reduction to sequential programs analysis
  - implemented as non-intrusive source transformation
  - lazy sequentialization for effective bug-hunting
    [Inverso et al., CAV'14]
- extension to WMMs…?
  - reduction to concurrent program analysis under SC
  - implemented as non-intrusive source transformation
    (based on shared memory abstractions)

Shared Memory Abstractions (SMA)

Practical perspective

- software-level API to encapsulate all shared memory operations:

```
read(v, t)       write(v, val, t)
create(f, t)     join(t', t)
lock(m, t)       unlock(m, t)
fence(t)         ...
```

thread id

shared memory
shared memory abstraction (API)
store buffer
store buffer
t_1 ... t_N
threads

- rewrite program over SMA:

```
x=y+z;   ==>   yl=read(y, t);
               zl=read(z, t);
               write(x, yl+zl, t);
```

- plug-in implementations for different memory semantics (SC, TSO, PSO)

Shared Memory Abstractions (SMA)

Theoretical perspective

- program $P = C \mid M$ is labelled transition system
  - labelled with / synchronized via signature of SMA Σ_{SMA}
  - C control-flow transition system (i.e., threads)
  - M SMA transition system
- L thread-asynchronous

  ⇔ all words α have same per-thread projection π(α)
- $L^{\#}$ thread-asynchronous closure
- $L_1 \sim L_2$ thread-wise equivalent

  ⇔ $\alpha \in L_1$ implies $\alpha' \in L_2$ s.t. $\pi(\alpha) = \pi(\alpha')$

⇒ thread-asynchronous and thread-wise equivalence preserve per-thread order but “factor out” interleavings

from each thread’s perspective: all words are the same

L_1 and L_2 contain same words (modulo per-thread projection)

Shared Memory Abstractions (SMA)

Theoretical perspective

- **Theorem:** If $L(M)$ is thread-asynchronous and $L(C_1) \sim L(C_2)$, then an error is reachable in $C_1 \mid M$ iff it is reachable in $C_2 \mid M$.

If M is thread-asynchronous, we can re-arrange thread interleavings without breaking the verification procedure.

- **Theorem:** If $L(M_1) = L(M_2)^\#$, then an error is reachable in $C \mid M_1$ iff it is reachable in $C \mid M_2$.

We can replace a memory model by a thread-asynchronous SMA without breaking the verification procedure.

(Bounded) Verification with SMAs

Guess memory unwinding

= data structure(s) to store sequence of writes over Σ_{SMA}

– naïve: array of (var, val, thread) triples

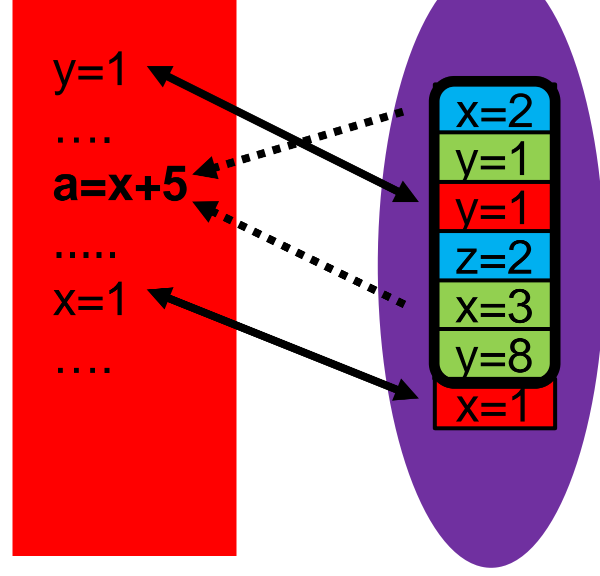
Assume-guarantee reasoning:

t_1		t_2
assumes: MU writes of other threads **guarantees**: its own MU writes	$\wedge$	**assumes**: MU writes of other threads **guarantees**: its own MU writes

(Bounded) Verification with SMAs

Check assume-guarantee condition for each thread:

- for each write:
  - get next write for thread id
  - check right variable
  - check right value
- ensure all writes are matched
- for each read:
  - pick value from write in right range

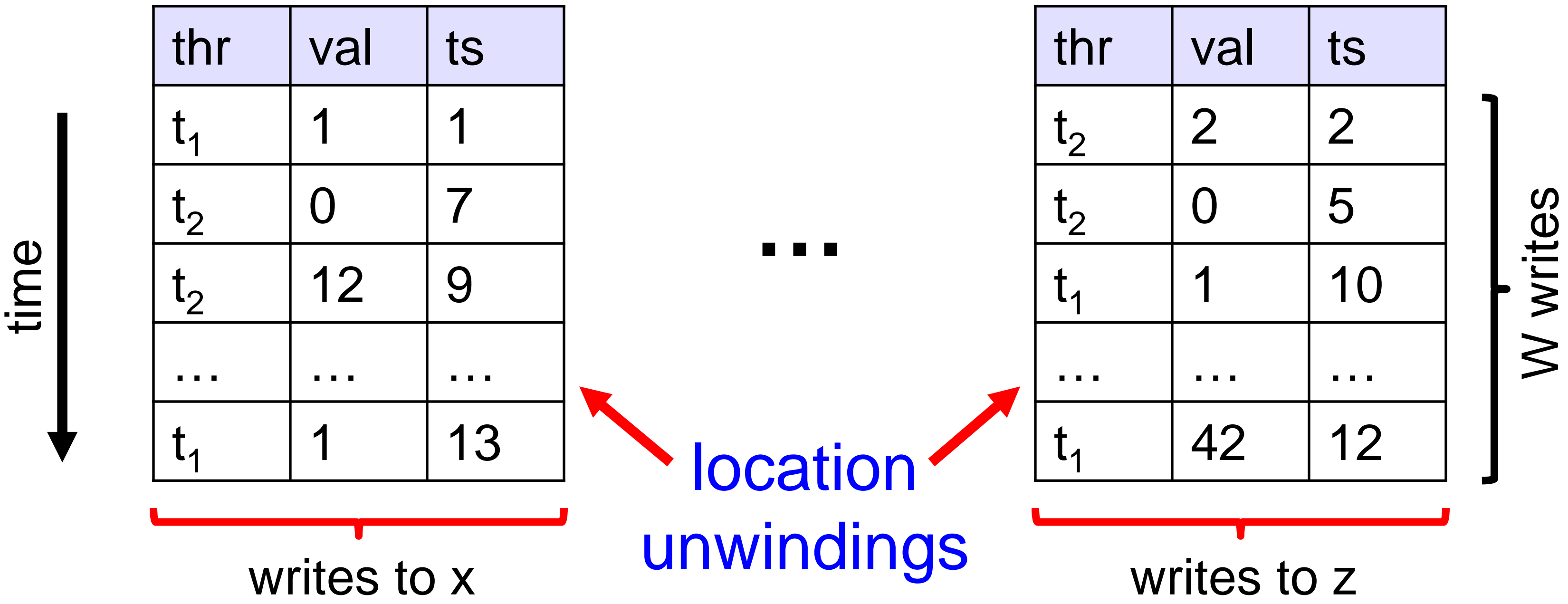


depends on memory model

Individual Memory-Location Unwinding (IMU)

Alternative representation of writes (for now for SC):

array of (thread, value, timestamp) triples per variable



thr	val	ts
t_1	1	1
t_2	0	7
t_2	12	9
…	…	…
t_1	1	13

thr	val	ts
t_2	2	2
t_2	0	5
t_1	1	10
…	…	…
t_1	42	12

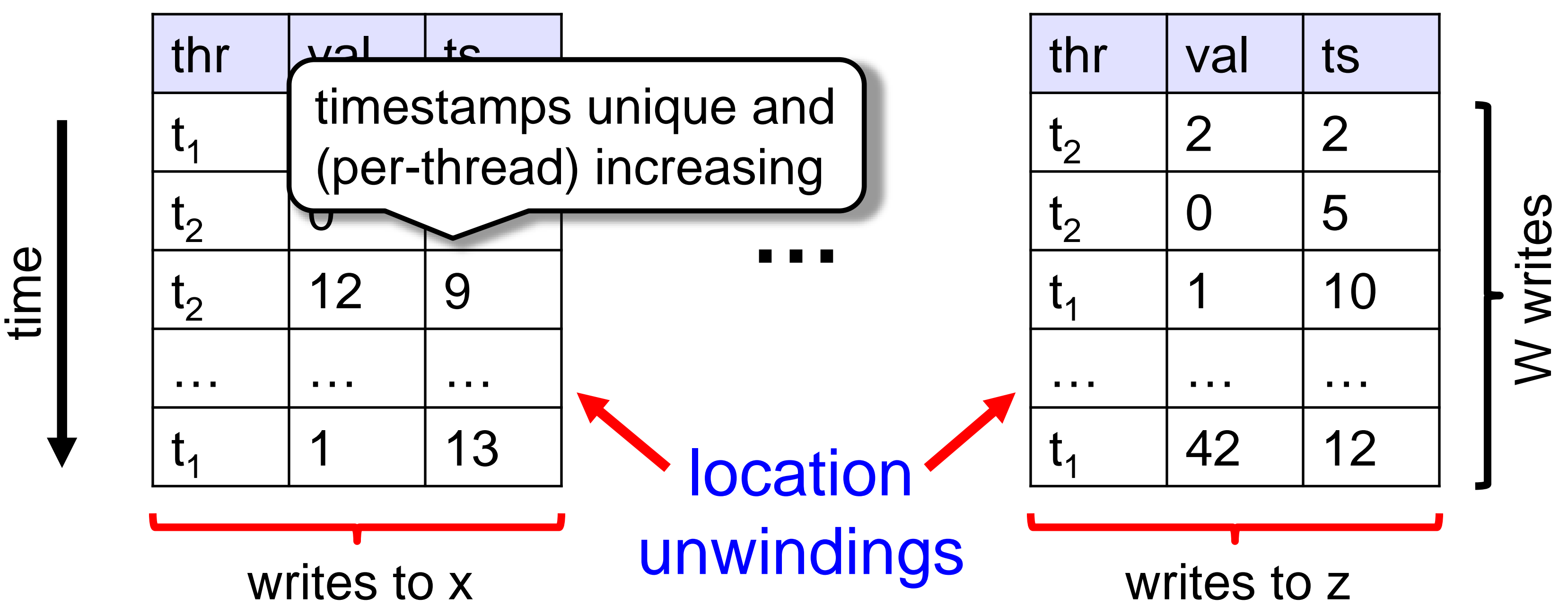
+ array to "link" writes of each thread in temporal order

thr_next_write[v][t][i] = { index of next write of t to v after i ; W (if no more writes) }

Individual Memory-Location Unwinding (IMU)

Alternative representation of writes (for now for SC):

array of (thread, value, timestamp) triples per variable



+ array to "link" writes of each thread in temporal order

thr_next_write[v][t][i] = { index of next write of t to v after i ; W (if no more writes) }

(Bounded) Verification with IMU-based SC SMAs

Implementation of `read` (see paper for `write`):

```
int read(uint v, uint t){
  if(is_terminated(t)) return 0;
  return (val[v][jump(v,t)]);
}

uint jump(uint v, uint t){
  uint j = *;  // non-deterministic choice
  uint k = thr_pos[v][t];
  assume( (j <= last_write[v])
       && (j <  thr_next_write[v][k][t])
       && (ts[v][j+1] > cur_ts[t]) );
  cur_ts[t] = (ts[v][j]>cur_ts[t]) ? ts[v][j] : cur_ts[t];
  return j;     // return if compatible with IMU
}
```

(Bounded) Verification with IMU-based SC SMAs

Implementation of `read` (see paper for `write`):

```
int read(uint v, uint t){
  if(is_terminated(t)) return 0;
  return (val[v][jump(v,t)]);
}

uint jump(uint v, uint t){
  uint j = *;  // non-deterministic choice
  uint k = thr_pos[v][t];
  assume( (j <= last_write[v])
       && (j <  thr_next_write[v][k][t])
       && (ts[v][j+1] > cur_ts[t]) );
  cur_ts[t] = (ts[v][j]>cur_ts[t]) ? ts[v][j] : cur_ts[t];
  return j;     // return if compatible with IMU
}
```

pick index in range and
return value from v-LU

(Bounded) Verification with IMU-based SC SMAs

Implementation of `read` (see paper for `write`):

```
int read(uint v, uint t){
  if(is_terminated(t)) return 0;
  return (val[v][jump(v,t)]);
}

uint jump(uint v, uint t){
  uint j = *;    // non-deterministic choice
  uint k = thr_pos[v][t];
  assume( (j <= last_write[v])
       && (j <  thr_next_write[v][k][t])
       && (ts[v][j+1] > cur_ts[t]) );
  cur_ts[t] = (ts[v][j]>cur_ts[t]) ? ts[v][j] : cur_ts[t];
  return j;      // return if compatible with IMU
}
```

pick index in range and return value from v-LU

current position of thread in v-LU (updated by write)

(Bounded) Verification with IMU-based SC SMAs

Implementation of `read` (see paper for `write`):

```
int read(uint v, uint t){
  if(is_terminated(t)) return 0;
  return (val[v][jump(v,t)]);
}

uint jump(uint v, uint t){
  uint j = *;      // non-deterministic choice
  uint k = thr_pos[v][t];
  assume( (j <= last_write[v])
       && (j <  thr_next_write[v][k][t])
       && (ts[v][j+1] > cur_ts[t]) );
  cur_ts[t] = (ts[v][j]>cur_ts[t]) ? ts[v][j] : cur_ts[t];
  return j;      // return if compatible with IMU
}
```

pick index in range and return value from v-LU

current position of thread in v-LU (updated by write)

current time stamp of t (updated by read and write)

(Bounded) Verification with IMU-based SC SMAs

Implementation of `read` (see paper for `write`):

```
int read(uint v, uint t){
  if(is_terminated(t)) return 0;
  return (val[v][jump(v,t)]);
}

uint jump(uint
  uint j = *;   //                        ce
  uint k = thr_pos[v]  ];
  assume( (j <= last_write[v])
       && (j <  thr_next_write[v][k][t])
       && (ts[v][j+1] > cur_ts[t]) );
  cur_ts[t] = (ts[v][j]>cur_ts[t]) ? ts[v][j] : cur_ts[t];
  return j;     // return if compatible with IMU
}
```

pick index in range and return value from v-LU

current position of thread in v-LU

can't read beyond last write to v (computed from guess)

current time stamp of t (updated by read and write)

(Bounded) Verification with IMU-based SC SMAs

Implementation of `read` (see paper for `write`):

```
int read(uint v, uint t){
  if(is_terminated(t)) return 0;
  return (val[v][jump(v,t)]);
}

uint jump(uint
  uint j = *;    //
  uint k = thr_pos[v][t];
  assume( (j <= last_write[v])
       && (j <  thr_next_write[v][k][t])
       && (ts[v][j+1] > cur_ts[t]) );
  cur_ts[t] = (ts[v][j]>cur_ts[t]) ? ts[v][j] : cur_ts[t];
  return j;    // return if compatible with IMU
}
```

pick index in range and return value from v-LU

current position of thread in v-LU

can't read beyond last write to v (co

can't read beyond next write by t to v (guessed at start)

current time stamp of t (updated by read and write)

(Bounded) Verification with IMU-based SC SMAs

Implementation of `read` (see paper for `write`):

```
int read(uint v, uint t){
  if(is_terminated(t)) return 0;
  return (val[v][jump(v,t)]);
}

uint jump(uint
  uint j = *;    //
  uint k = thr_pos[v][t];
  assume( (j <= last_write[v][t])
       && (j <  thr_next_write[v][
       && (ts[v][j+1] > cur_ts[t]) );
  cur_ts[t] = (ts[v][j]>cur_ts[t]) ? ts[v][j] : cur_ts[t];
  return j;     // return if compatible with IMU
}
```

pick index in range and return value from v-LU

current position of thread in v-LU

can't read beyond last write to v (co

can't read beyond next write by t to v

next write by anyone to v must be after current time stamp of t

current time stamp of t (updated by read and write)

(Bounded) Verification with IMU-based SC SMAs

Implementation of `read` (see paper for `write`):

```
int read(uint v, uint t){
  if(is_terminated(t)) return 0;
  return (val[v][jump(v,t)]);
}

uint jump(ui
  uint j = *;    /
  uint k = thr_pos[v]
  assume( (j <= last_write[
       && (j <  thr_next_write[v][
       && (ts[v][j+1] > cur_ts[t]) );
  cur_ts[t] = (ts[v][j]>cur_ts[t]) ? ts[v][j] : cur_ts[t];
  return j;    // return if compatible with IMU
}
```

pick index in range and return value from v-LU

current position of thread in v-LU

can't read beyond last write to v (co

can't read beyond next write by t to v

next write by anyone to v must be after current time stamp of t

current time stamp of t (updated by read and write)

pull current time stamp up to picked write

IMU implementation for TSO

Optimized representation of writes for TSO/PSO:

array of (thread, value, timestamp, timestamp) quadruples per variable

thr	val	bts	ts
t_1	1	1	1
t_2	0	3	7
t_2	12	5	9
…	…	…	
t_1	1	10	13

writes to x

…

thr	val	bts	ts
t_2	2	2	2
t_2	0	4	5
t_1	1	8	10
…	…	…	
t_1	42	10	12

writes to z

IMU implementation for TSO

Optimized representation of writes for TSO/PSO:

array of [illegible] mestamp, time[illegible] s
per vari[illegible]

timestamp of write into buffer

timestamp of update into shared memory

time ↓

thr	val	bts	ts
t_1	1	1	1
t_2	0	3	7
t_2	12	5	9
…	…	…	
t_1	1	10	13

writes to x

…

thr	val	bts	ts
t_2	2	2	2
t_2	0	4	5
t_1	1	8	10
[illegible]	[illegible]	…	
[illegible]	[illegible]	10	12

W writes

writes to z

buffer writes must happen before updates

⇒ no explicit representation of buffers required!

(Bounded) Verification with IMU-based TSO SMAs

Implementation of `read` (see paper for `write` and `fence`):

```
int read(uint v, uint t){
  if(is_terminated(t)) return 0;
  uint j = *;
  uint k = thr_pos[v][t];
  uint nxt_write = thr_next_write[v][k][t];
  uint fst_write = thr_next_write[v][0][t];
  assume( (j >= cur_ts[t])
       && (j <  bts[v][nxt_write]) );
  cur_ts[t] = j;
  if( fst_write <= k && ts[v][k] > cur_ts[t] )
    return val[v][k];
  return read_SC(v,t);
}
```

(Bounded) Verification with IMU-based TSO SMAs

Implementation of `read` (see paper for `write` and `fence`):

```
int read(uint v, uint t){
  if(is_terminated(t)) return 0;
  uint j = *;
  uint k = thr_pos[v][t];
  uint nxt_write = thr_next_
  uint fst_write = thr_next_
  assume( (j >= cur_ts[t])
       && (j <  bts[v][nxt_write]) );
  cur_ts[t] = j;
  if( fst_write <= k && ts[v][k] > cur_ts[t] )
    return val[v][k];
  return read_SC(v,t);
}
```

randomly increase t’s timestamp
(but not beyond next write to buffer)
to model buffer flushing into memory

(Bounded) Verification with IMU-based TSO SMAs

Implementation of `read` (see paper for `write` and `fence`):

```
int read(uint v, uint t){
  if(is_terminated(t)) return 0;
  uint j = *;
  uint k = thr_pos[v][t];
  uint nxt_write = thr_next_
  uint fst_write = thr_next_write      ][t];
  assume( (j >= cur_ts[t])
       && (j <  bts[v][nxt_write]) );
  cur_ts[t] = j;
  if( fst_write <= k && ts[v][k] > cur_ts[t] )
    return val[v][k];
  return read_SC(v,t);
}
```

randomly increase t’s timestamp
(but not beyond next write to buffer)
to model buffer flushing into memory

return buffered value
if not yet flushed

(Bounded) Verification with IMU-based TSO SMAs

Implementation of `read` (see paper for `write` and `fence`):

```
int read(uint v, uint t){
  if(is_terminated(t)) return 0;
  uint j = *;
  uint k = thr_pos[v][t];
  uint nxt_write = thr_next_
  uint fst_write = thr_next_write        ][t];
  assume( (j >= cur_ts[t])
       && (j <  bts[v][nxt_write]) );
  cur_ts[t] = j;
  if( fst_write <= k && ts[v][k] > cur_ts[t] )
    return val[v][k];
  return read_SC(v,t);
}
```

randomly increase t’s timestamp (but not beyond next write to buffer) to model buffer flushing into memory

return buffered value if not yet flushed

else perform SC read

(Bounded) Verification with IMU-based PSO SMAs

Changes required to write, to reflect FIFO per variable instead of entire buffer

$\Rightarrow$ SMA implementation becomes simpler!

(See paper for details)

Experimental Evaluation

Compared IMU-SMA implementation in CSeq with

- Lazy-SMA (also implemented CSeq) [Tomasco et al., FMCAD'16]
- CBMC (uses formula representation) [Alglave et al., CAV'13]
- Nidhugg (stateless MC with DPOR) [Abdulla et al., TACAS'15]

Benchmarks:

- mutual exclusion (really simple…)
- SV-COMP concurrency (simple)
- industrial-ish code (medium)
- safestack (concurrent data structure, really hard…)

Experimental Evaluation - Results

	l.o.c.	parameters: unwind	W	U	M	bitwidth	TSO runtime (s): bug?	files	IMU-CSeq	LazySMA	CBMC	NIDHUGG	PSO runtime (s): bug?	files	IMU-CSeq	LazySMA	CBMC	NIDHUGG
dekker	52	1	2	0	0	5	•	1	0.76	0.77	0.29	**0.04**	•	1	0.76	0.75	0.25	**0.05**
lamport	78	1	2	0	0	5	•	1	0.97	0.88	0.31	**0.05**	•	1	0.97	0.88	0.29	**0.05**
peterson	40	1	3	0	0	5	•	1	0.67	0.66	0.26	**0.04**	•	1	0.68	0.65	0.25	**0.04**
szymanski	57	1	3	0	0	5	•	1	0.84	0.81	0.34	**0.07**	•	1	0.84	0.80	0.32	**0.04**
fib_longer_unsafe	30	6	7	0	0	10	•	1	**2.10**	6.47	8.19	94.84	•	1	2.50	6.51	**1.69**	135.45
fib_longer_safe	30	6	7	0	0	10		1	**4.75**	9.78	22.5	t.o.		1	**3.90**	8.82	31.8	t.o.
pgsql	47	1	2	0	0	5		1	1.92	2.03	**0.03**	0.07	•	1	0.69	0.65	0.22	**0.04**
parker	110	1	2	0	0	5	•	1	1.22	1.68	0.31	**0.05**	•	1	1.21	2.19	0.28	**0.05**
stack_unsafe	110	2	2	1	2	5	•	1	1.46	1.50	0.41	**0.05**	•	1	1.44	1.49	0.35	**0.05**
litmus_safe	-	1	6	1	0	10		5526	1.20	1.26	**0.17**	2.35		4835	1.06	1.22	**0.15**	6.65
litmus_unsafe	-	1	6	1	0	10	•	277	1.67	1.27	**0.16**	3.86	•	968	1.28	1.26	**0.12**	1.58
safestack	83	3	10	7	2	5	•	1	**207.4**	1474.6	t.o.	t.o.	•	1	**1013.3**	1207.3	t.o.	t.o.

- Nidhugg best for most simple benchmarks
  - except litmus (submitted by CBMC team)
  - SMA implementations pay small performance penalty (mostly from source-to-source transformations)
- SMA implementation shine on harder examples
  - IMU-SMA (slightly) outperforms Lazy-SMA

Conclusions

- first source-level handling of TSO and PSO suitable for eager SC backends
  - complements Lazy-SMA
  - more evidence that sequentialization and SMAs are good frameworks
- implementation is competitive

Future Work

- extension to POWER (and other WMMs)
- application to message passing
  - write → send, read → receive
- application to other languages (Java?)

Shameless plug…

ICTAC 2018 – International Colloquium on Theoretical Aspects of Computing

Stellenbosch, early/mid October 2018