Lazy Sequentialization for TSO and PSO via Shared Memory Abstractions

Ermenegildo Tomasco University of Southampton, UK
Truc Lam Nguyen University of Southampton, UK
Omar Inverso Gran Sasso Science Institute, Italy
Bernd Fischer Stellenbosch University, South Africa
Salvatore La Torre Università di Salerno, Italy
Gennaro Parlato University of Southampton, UK

UNIVERSITY OF Southampton

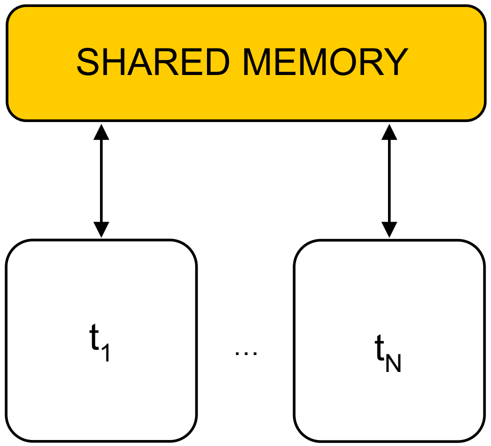
GRAN SASSO SCIENCE INSTITUTE
CENTER FOR ADVANCED STUDIES
Istituto Nazionale di Fisica Nucleare

UNIVERSITEIT STELLENBOSCH UNIVERSITY

UNIVERSITÀ DEGLI STUDI DI SALERNO

FMCAD 2016, Mountain View, CA, USA

Relaxed Memory Consistency



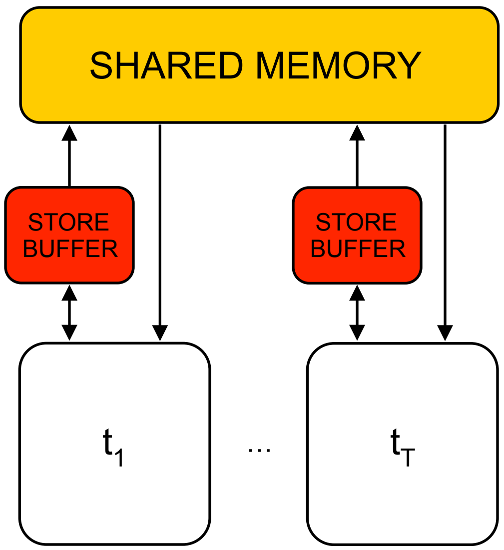
sequential consistency (SC)

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

weak memory models (WMMs)

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

Relaxed Memory Consistency



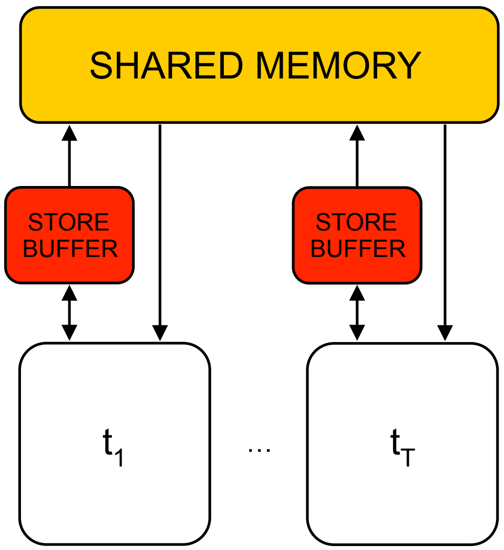
total store order (TSO)

- writes executed in their order for each thread
- reads may overtake writes

partial store order (PSO)

- writes to the same location executed in their order for each thread
- writes to different locations may be reordered
- reads may overtake writes

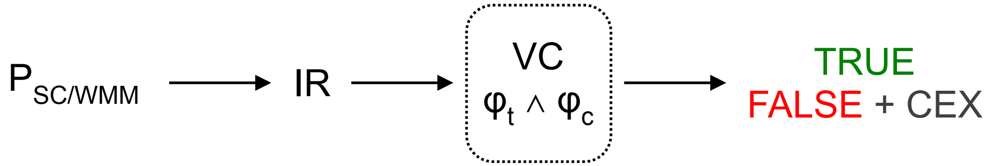
Relaxed Memory Consistency



limitations of testing

- generally ineffective for rare concurrency errors
- cannot control additional nondeterminism introduced by WMMs
- need to be complemented with symbolic analysis

Symbolic Bug Finding: BMC

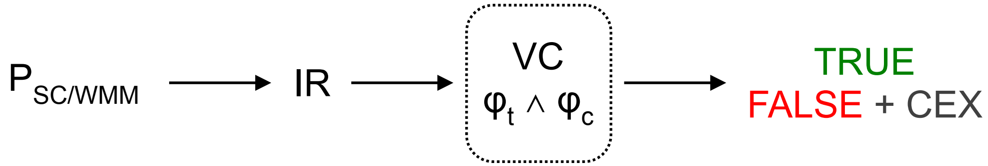


concurrency handling at formula level

- encode threads separately
- add φ_c to capture thread interleaving

[Sinha, Wang – POPL 2011]

Symbolic Bug Finding: BMC



concurrency handling at formula level

- encode threads separately
- add φ_c to capture thread interleaving

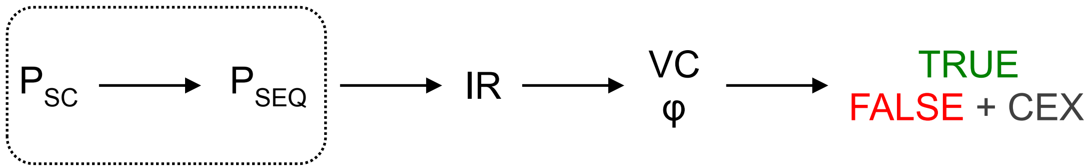
[Sinha, Wang – POPL 2011]

extension to WMMs is natural

- change φ_c to capture extra interactions due to weaker consistency

[Alglave, Kroening, Tautschnig – CAV 2013]

Symbolic Bug Finding:
Lazy Sequentialization + BMC

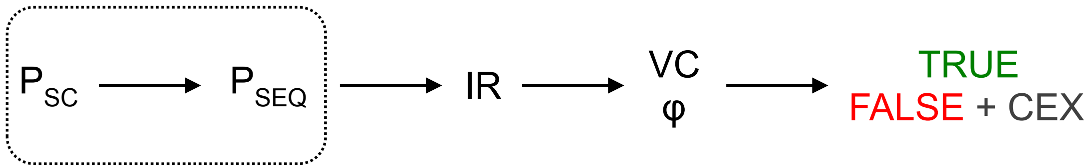


concurrency handling at code level

- reduction to sequential programs analysis
- implemented as source transformation
- lazy sequentialization tailored to BMC for effective in bug-hunting

[Inverso, Tomasco, Fischer, La Torre, Parlato – CAV 2014]

Symbolic Bug Finding: Lazy Sequentialization + BMC



concurrency handling at code level

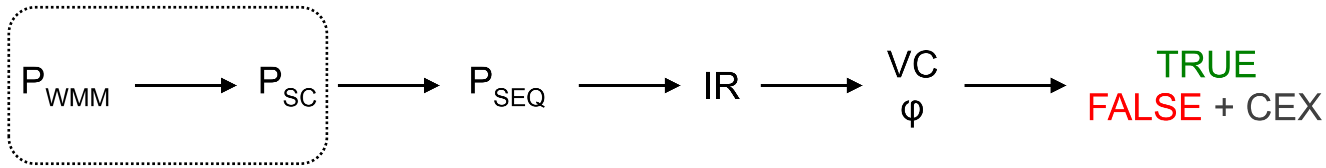
- reduction to sequential programs analysis
- implemented as source transformation
- lazy sequentialization tailored to BMC for effective in bug-hunting

[Inverso, Tomasco, Fischer, La Torre, Parlato – CAV 2014]

how to extend to WMMs?

how does it compare?

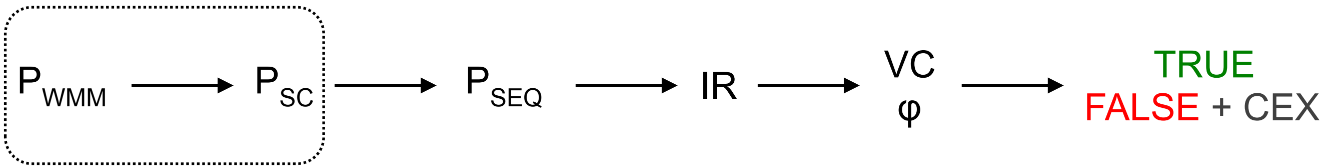
Extending Lazy Sequentialization to TSO and PSO



how to extend to WMMs?

- reduction to concurrent program analysis under SC
- again, implemented as source transformation

Extending Lazy Sequentialization to TSO and PSO



how to extend to WMMs?

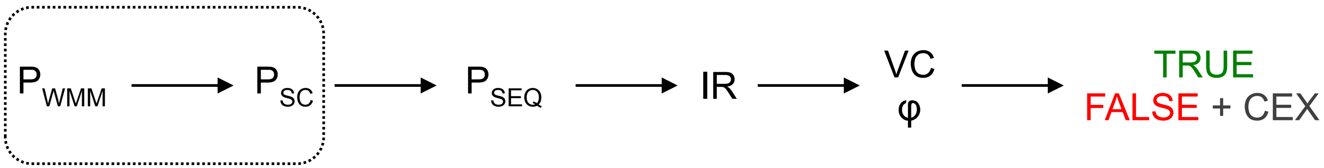
- reduction to concurrent program analysis under SC
- again, implemented as source transformation
  - replace shared memory access with explicit function calls to SMA API:

    `read(v,t)`, `write(v,val,t)`

    `lock(m,t)`, `unlock(m,t)`, `fence(t)`, …

    example: `x=y+3` is changed to `write(x,read(y)+3)`

Extending Lazy Sequentialization to TSO and PSO



how to extend to WMMs?

- reduction to concurrent program analysis under SC
- again, implemented as source transformation
  - replace shared memory access with explicit function calls to SMA API:

    `read(v,t)`, `write(v,val,t)`

    `lock(m,t)`, `unlock(m,t)`, `fence(t)`, …

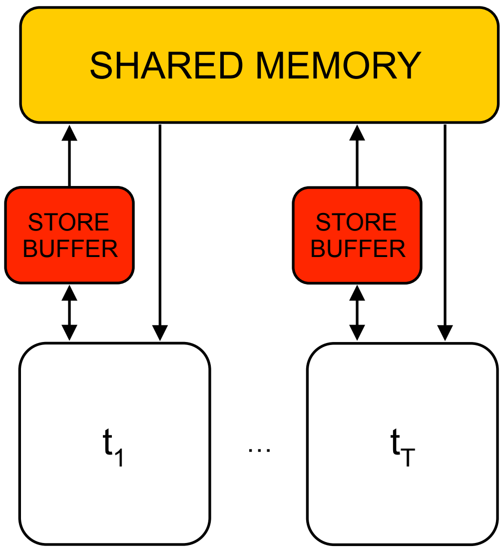
    example: `x=y+3` is changed to `write(x,read(y)+3)`
  - plug in implementation for specific semantics

    TSO-SMA - simple implementation

    eTSO-SMA - efficient implementation

    PSO-SMA - extension to PSO

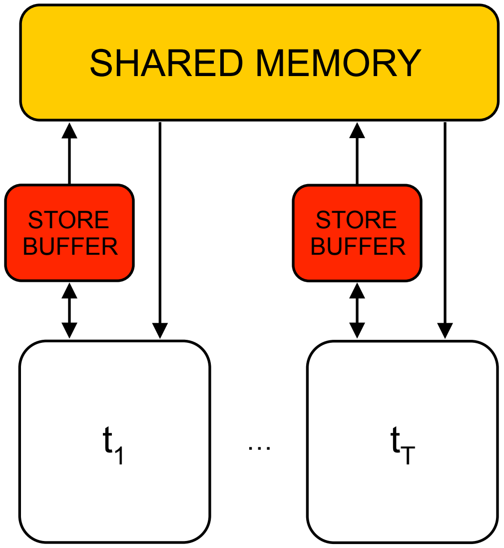
TSO-SMA



simple simulation of the store buffer

- introduce <u>one array for each thread</u>
- `read(v,t)`
  - look up buffer for pending writes
  - fetch from memory
- `write(v,val,t)`
  - update store buffer
  - inject nondeterministic memory flush

TSO-SMA

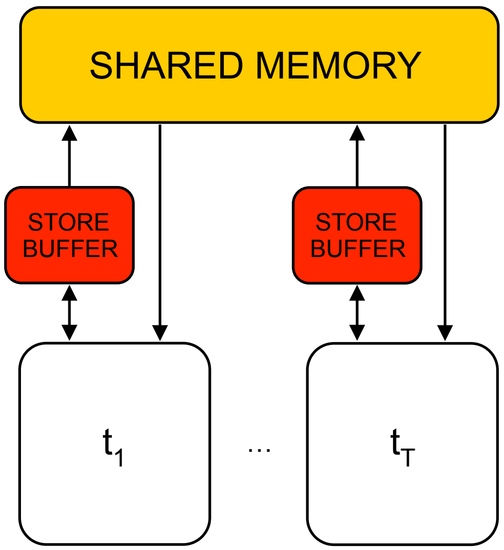


simple simulation of the store buffer

- introduce <u>one array for each thread</u>
- `read(v,t)`
  - look up buffer for pending writes
  - fetch from memory
- `write(v,val,t)`
  - update store buffer
  - inject nondeterministic memory flush

formula size depends on store buffer size

TSO-SMA



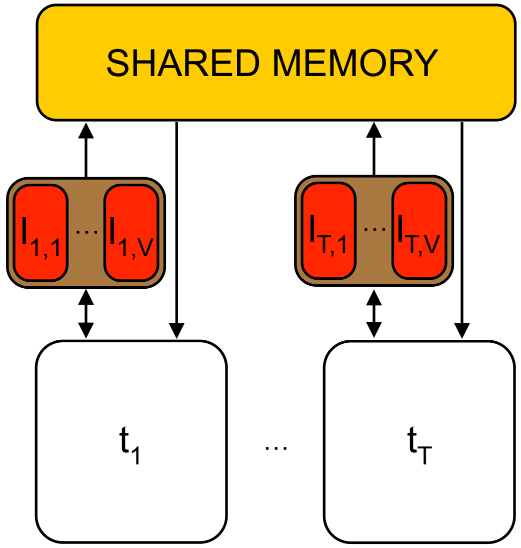
simple simulation of the store buffer

- introduce one array for each thread
- `read(v,t)`
  - look up buffer for pending writes
  - fetch from memory
- `write(v,val,t)`
  - update store buffer
  - inject nondeterministic memory flush

formula size depends on store buffer size

formula size proportional to
no. memory accesses
no. of store buffers
max no. of elems in the buffer

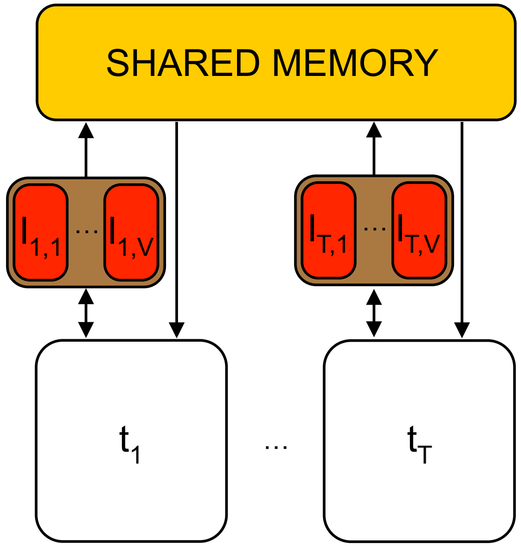
eTSO-SMA



efficient simulation of the store buffer

- introduce <u>one list for each shared variable and thread</u>
- use global clock and timestamp memory writes
- `read(v,t)`
  - buffer look up, return value from latest pending write
  - return value from latest expired write
- `write(v,val,t)`
  - guess timestamp, enforce non-decreasing order
  - update buffer

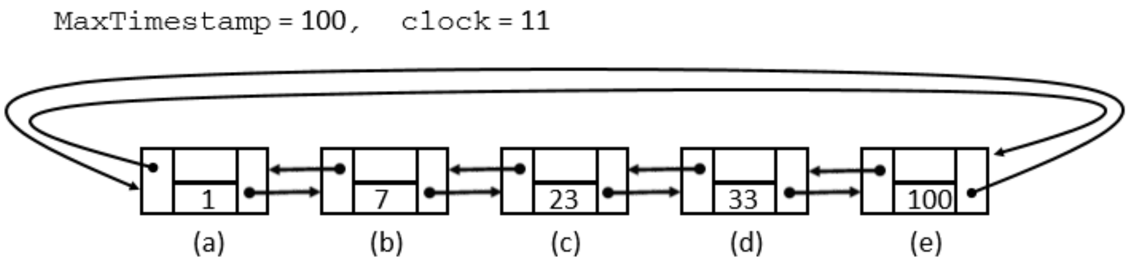
eTSO-SMA

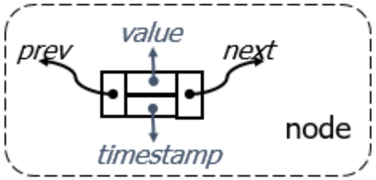


efficient simulation of the store buffer

- introduce <u>one list for each shared variable and thread</u>
- use global clock and timestamp memory writes
- `read(v,t)` constant size
  - buffer look up, return value from latest pending write
  - return value from latest expired write
- `write(v,val,t)` constant size
  - guess timestamp, enforce non-decreasing order
  - update buffer

Variable Write Lists (T-CDLL)



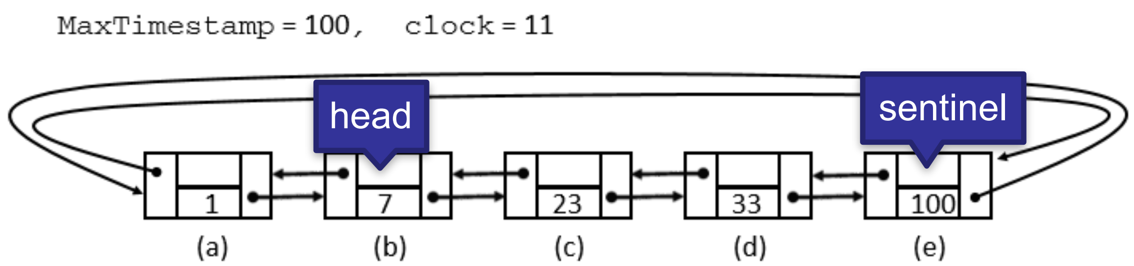


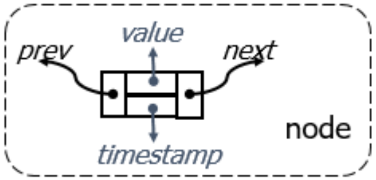
- store pairs *(value,timestamp)*
- clock determines expired nodes
- expired nodes not removed

special nodes

- **sentinel node**
  has max *timestamp*
  does not correspond to any actual write
- **head**
  only node to contain an expired write
  followed by a non-expired write

Variable Write Lists (T-CDLL)



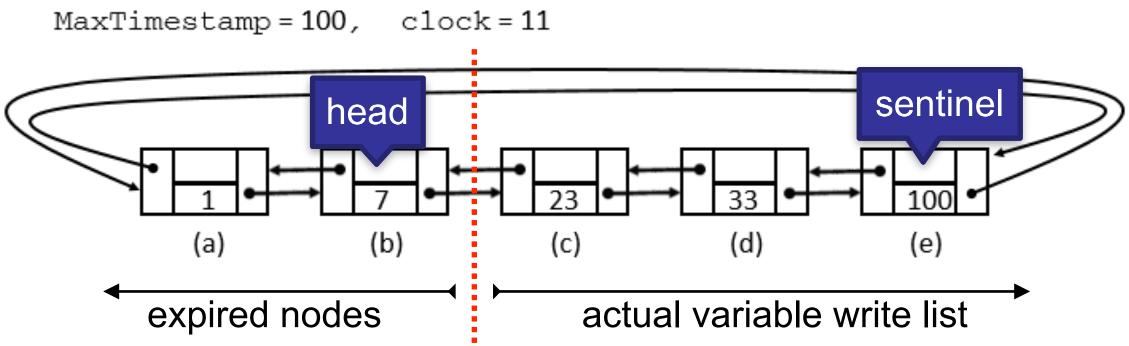


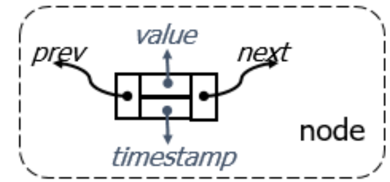
- store pairs *(value,timestamp)*
- clock determines expired nodes
- expired nodes not removed

special nodes

- **sentinel node**
  has max *timestamp*
  does not correspond to any actual write
- **head**
  only node to contain an expired write
  followed by a non-expired write

Variable Write Lists (T-CDLL)





- store pairs *(value,timestamp)*
- clock determines expired nodes
- expired nodes not removed

special nodes

- **sentinel node**
  has max *timestamp*
  does not correspond to any actual write
- **head**
  only node to contain an expired write
  followed by a non-expired write

Auxiliary Data Structures

parameters

`T` max no. of threads

`V` max no. of tracked locations (or write lists)

`N` max no. of nodes for each variable write list

`K` max timestamp

variables

```
int clock;
```

- variable write lists

```
int value[V][N+1],
    tstamp[V][N+1],
    prev[V][N+1],
    next[V][N+1];
```

- last values and timestamps

```
int last_value[V][T],
    last_tstamp[V][T];
```

- max timestamp so far

```
int max_tstamp[T];
```

eTSO-SMA: read operation

```
int clock_update() {
  int tmp = *;
  assume(clock <= tmp && tmp <= K);
  clock = tmp;
}

int read(int v, int t) {
  clock_update();

  if (last_tstamp[v][t] > clock)
    return last_value[v][t];

  int node = *;
  assume(node < N &&
         tstamp[v][node] <= clock &&
         tstamp[v][next[v][node]] > clock);
  return value[v][node];
}
```

eTSO-SMA: read operation

```
int clock_update() {
  int tmp = *;
  assume(clock <= tmp && tmp <= K);
  clock = tmp;
}

int read(int v, int t) {
  clock_update();

  if (last_tstamp[v][t] > clock)
    return last_value[v][t];

  int node = *;
  assume(node < N &&
         tstamp[v][node] <= clock &&
         tstamp[v][next[v][node]] > clock);
  return value[v][node];
}
```

clock follows non-decreasing order

eTSO-SMA: read operation

```
int clock_update() {
  int tmp = *;
  assume(clock <= tmp && tmp <= K);
  clock = tmp;
}
```

clock follows non-decreasing order

```
int read(int v, int t) {
  clock_update();

  if (last_tstamp[v][t] > clock)
    return last_value[v][t];

  int node = *;
  assume(node < N &&
         tstamp[v][node] <= clock &&
         tstamp[v][next[v][node]] > clock);
  return value[v][node];
}
```

if the last write by **t** on **v** has not expired, return the corresponding value

eTSO-SMA: read operation

```
int clock_update() {
  int tmp = *;
  assume(clock <= tmp && tmp <= K);
  clock = tmp;
}
```

clock follows non-decreasing order

```
int read(int v, int t) {
  clock_update();

  if (last_tstamp[v][t] > clock)
    return last_value[v][t];

  int node = *;
  assume(node < N &&
         tstamp[v][node] <= clock &&
         tstamp[v][next[v][node]] > clock);
  return value[v][node];
}
```

if the last write by **t** on **v** has not expired, return the corresponding value

return the value from the latest expired write, which is guaranteed to exist and correspond to the value of **v** in the memory

eTSO-SMA: read operation

```
int clock_update() {
  int tmp = *;
  assume(clock <= tmp && tmp <= K);
  clock = tmp;
}
```

clock follows non-decreasing order

```
int read(int v, int t) {
  clock_update();

  if (last_tstamp[v][t] > clock)
    return last_value[v][t];

  int node = *;
  assume(node < N &&
         tstamp[v][node] <= clock &&
         tstamp[v][next[v][node]] > clock);
  return value[v][node];
}
```

if the last write by **t** on **v** has not expired, return the corresponding value

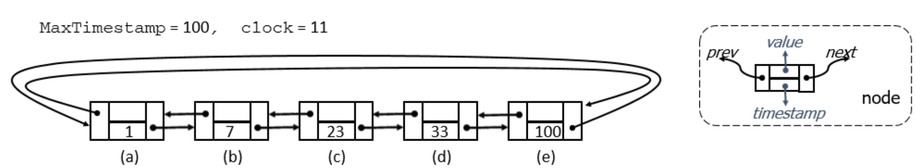
return the value from the latest expired write, which is guaranteed to correspond to the value of v

representation of the memory no longer needed

eTSO-SMA: write operation

select expired node with min timestamp for the new write

```
int write(int v, int t) {
  clock_update();
  int node = next[v][N];
  assume(tstamp[v][next[v][node]] <= clock);
  next[v][N] = next[v][node];
  prev[v][next[v][N]] = N;

  int succ = *;
  assume(succ <= N && tstamp[v][succ] > clock);
  int pred = prev[v][succ];

  int ts = *;
  assume(ts >= clock && ts >= max_tstamp[t]);
  assume(ts >= tstamp[v][pred] && ts < tstamp[v][succ]);

  value[v][node] = val;
  tstamp[v][t] = ts;
  …
  last_tstamp[v][t] = ts;
  last_value[v][t] = val;
  max_tstamp[t] = ts;
}
```

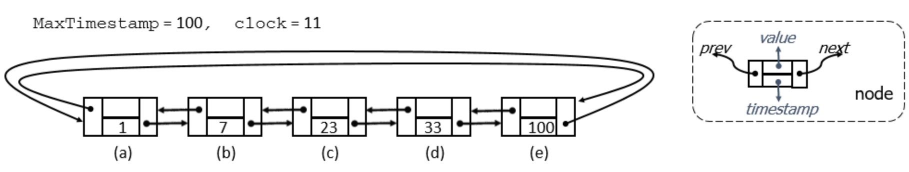


eTSO-SMA: write operation

```
int write(int v, int t) {
  clock_update();
  int node = next[v][N];
  assume(tstamp[v][next[v][node]] <= clock);
  next[v][N] = next[v][node];
  prev[v][next[v][N]] = N;

  int succ = *;
  assume(succ <= N && tstamp[v][succ] > clock);
  int pred = prev[v][succ];

  int ts = *;
  assume(ts >= clock && ts >= max_tstamp[t]);
  assume(ts >= tstamp[v][pred] && ts < tstamp[v][succ]);

  value[v][node] = val;
  tstamp[v][t] = ts;
  …
  last_tstamp[v][t] = ts;
  last_value[v][t] = val;
  max_tstamp[t] = ts;
}
```

select expired node with min timestamp for the new write

position the new node by nondeterministically selecting its successor among the non-expired nodes



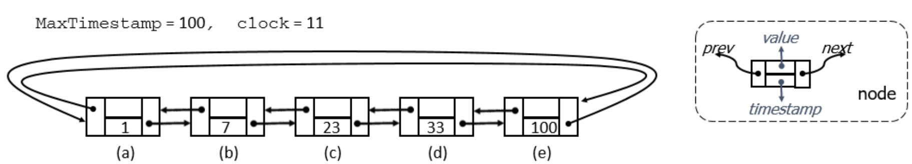
eTSO-SMA: write operation

```
int write(int v, int t) {
  clock_update();
  int node = next[v][N];
  assume(tstamp[v][next[v][node]] <= clock);
  next[v][N] = next[v][node];
  prev[v][next[v][N]] = N;

  int succ = *;
  assume(succ <= N && tstamp[v][succ] > clock);
  int pred = prev[v][succ];

  int ts = *;
  assume(ts >= clock && ts >= max_tstamp[t]);
  assume(ts >= tstamp[v][pred] && ts < tstamp[v][succ]);

  value[v][node] = val;
  tstamp[v][t] = ts;
  …
  last_tstamp[v][t] = ts;
  last_value[v][t] = val;
  max_tstamp[t] = ts;
}
```

select expired node with min timestamp for the new write

position the new node by nondeterministically selecting its successor among the non-expired nodes

guess suitable timestamp, must respect non-decreasing order



eTSO-SMA: write operation

```
int write(int v, int t) {
  clock_update();
  int node = next[v][N];
  assume(tstamp[v][next[v][node]] <= clock);
  next[v][N] = next[v][node];
  prev[v][next[v][N]] = N;

  int succ = *;
  assume(succ <= N && tstamp[v][succ] > clock);
  int pred = prev[v][succ];

  int ts = *;
  assume(ts >= clock && ts >= max_tstamp[t]);
  assume(ts >= tstamp[v][pred] && ts < tstamp[v][succ]);

  value[v][node] = val;
  tstamp[v][t] = ts;
  …
  last_tstamp[v][t] = ts;
  last_value[v][t] = val;
  max_tstamp[t] = ts;
}
```

select expired node with min timestamp for the new write

position the new node by nondeterministically selecting its successor among the non-expired nodes

guess suitable timestamp, must respect non-decreasing order

update variable write list and auxiliary variables

extension to PSO

```
int write(int v, int t) {
  clock_update();
  int node = next[v][N];
  assume(tstamp[v][next[v][node]] <= clock);
  next[v][N] = next[v][node];
  prev[v][next[v][N]] = N;

  int succ = *;
  assume(succ <= N && tstamp[v][suc
  int pred = prev[v][succ];

  int ts = *;
  assume(ts >= clock && ts >= max_tstamp[t]);
  assume(ts >= tstamp[v][pred] && ts < tstamp[v][succ]);

  value[v][node] = val;
  tstamp[v][t] = ts;
  …
  last_tstamp[v][t] = ts;
  last_value[v][t] = val;
  max_tstamp[t] = ts;
}
```

write to different variables may be reordered, guessed timestamps no longer need to be the maximum over all variables, but the maximum for the relevant variable:

```
ts >= last_tstamp[t][v]
```

extension to PSO

```
int write(int v, int t) {
  clock_update();
  int node = next[v][N];
  assume(tstamp[v][next[v][node]] <= clock);
  next[v][N] = next[v][node];
  prev[v][next[v][N]] = N;

  int succ = *;
  assume(succ <= N && tstamp[v][suc
  int pred = prev[v][succ];

  int ts = *;
  assume(ts >= clock && ts >= max_tstamp[t]);
  assume(ts >= tstamp[v][pred] && ts < tstamp[v][succ]);

  value[v][node] = val;
  tstamp[v][t] = ts;
  …
  last_tstamp[v][t] = t
  last_value[v][t] = va
  max_tstamp[t] = ts;
}
```

write to different variables may be reordered, guessed timestamps no longer need to be the maximum over all variables, but the maximum for the relevant variable:

```
ts >= last_tstamp[t][v]
```

guessed timestamps may be smaller than the max timestamp:

```
max_tstamp[t] =
max(max_tstamp[t],ts)
```

Experimental Evaluation: common benchmarks

	bug?	unwind	qsize (N)	naddr	nmalloc	bitwidth	rounds	maxclock (K)	TSO runtime (s) LazySMA	TSO CBMC	TSO NIDHUGG	PSO runtime (s) LazySMA	PSO CBMC	PSO NIDHUGG
dekker	●	1	2	0	0	4	2	2	0.77	0.29	**0.04**	0.75	0.25	**0.05**
lamport	●	1	2	0	0	4	2	2	0.88	0.31	**0.05**	0.88	0.29	**0.05**
peterson	●	1	3	0	0	4	2	2	0.66	0.26	**0.04**	0.65	0.25	**0.04**
szymanski	●	1	3	0	0	4	2	3	0.81	0.34	**0.07**	0.80	0.32	**0.04**
fib_longer_unsafe	●	6	2	0	0	10	6	2	**6.47**	8.19	94.84	6.51	**1.69**	135.45
fib_longer_safe		6	2	0	0	10	6	2	**9.78**	22.5	t.o.	**8.82**	31.8	t.o.
parker	●	1	2	0	0	4	2	3	1.68	0.31	**0.05**	2.19	0.28	**0.05**
stack_unsafe	●	2	2	1	2	5	2	2	1.50	0.41	**0.05**	1.49	0.35	**0.05**
litmus_safe (avg)		5	2	0	0	10	2	20	1.26	**0.17**	2.35	1.22	**0.15**	6.65
litmus_unsafe (avg)	●	5	2	0	0	10	2	20	1.27	**0.16**	3.86	1.26	**0.12**	1.58

timeout = 600s

transformation overhead shows on small programs

Experimental Evaluation: common benchmarks

	bug?	unwind	qsize (N)	naddr	nmalloc	bitwidth	rounds	maxclock (K)	TSO runtime (s) LazySMA	TSO CBMC	TSO NIDHUGG	PSO runtime (s) LazySMA	PSO CBMC	PSO NIDHUGG
dekker	●	1	2	0	0	4	2	2	0.77	0.29	**0.04**	0.75	0.25	**0.05**
lamport	●	1	2	0	0	4	2	2	0.88	0.31	**0.05**	0.88	0.29	**0.05**
peterson	●	1	3	0	0	4	2	2	0.66	0.26	**0.04**	0.65	0.25	**0.04**
szymanski	●	1	3	0	0	4	2	3	0.81	0.34	**0.07**	0.80	0.32	**0.04**
fib_longer_unsafe	●	6	2	0	0	10	6	2	**6.47**	8.19	94.84	6.51	**1.69**	135.45
fib_longer_safe		6	2	0	0	10	6	2	**9.78**	22.5	t.o.	**8.82**	31.8	t.o.
parker	●	1	2	0	0	4	2	3	1.68	0.31	**0.05**	2.19	0.28	**0.05**
stack_unsafe	●	2	2	1	2	5	2	2	1.50	0.41	**0.05**	1.49	0.35	**0.05**
litmus_safe (avg)		5	2	0	0	10	2	20	1.26	**0.17**	2.35	1.22	**0.15**	6.65
litmus_unsafe (avg)	●	5	2	0	0	10	2	20	1.27	**0.16**	3.86	1.26	**0.12**	1.58

timeout = 600s

competitive on twisted interleavings

Experimental Evaluation: common benchmarks

	bug?	unwind	qsize (N)	naddr	nmalloc	bitwidth	rounds	maxclock (K)	TSO runtime (s) LazySMA	TSO CBMC	TSO NIDHUGG	PSO runtime (s) LazySMA	PSO CBMC	PSO NIDHUGG
dekker	●	1	2	0	0	4	2	2	0.77	0.29	**0.04**	0.75	0.25	**0.05**
lamport	●	1	2	0	0	4	2	2	0.88	0.31	**0.05**	0.88	0.29	**0.05**
peterson	●	1	3	0	0	4	2	2	0.66	0.26	**0.04**	0.65	0.25	**0.04**
szymanski	●	1	3	0	0	4	2	3	0.81	0.34	**0.07**	0.80	0.32	**0.04**
fib_longer_unsafe	●	6	2	0	0	10	6	2	**6.47**	8.19	94.84	6.51	**1.69**	135.45
fib_longer_safe		6	2	0	0	10	6	2	**9.78**	22.5	t.o.	**8.82**	31.8	t.o.
parker	●	1	2	0	0	4	2	3	1.68	0.31	**0.05**	2.19	0.28	**0.05**
stack_unsafe	●	2	2	1	2	5	2	2	1.50	0.41	**0.05**	1.49	0.35	**0.05**
litmus_safe (avg)		5	2	0	0	10	2	20	1.26	**0.17**	2.35	1.22	**0.15**	6.65
litmus_unsafe (avg)	●	5	2	0	0	10	2	20	1.27	**0.16**	3.86	1.26	**0.12**	1.58

timeout = 600s

slower

Experimental Evaluation: common benchmarks

	bug?	parameters: unwind	qsize (N)	naddr	nmalloc	bitwidth	rounds	maxclock (K)	TSO runtime (*s*): LazySMA	CBMC	NIDHUGG	PSO runtime (*s*): LazySMA	CBMC	NIDHUGG
dekker	●	1	2	0	0	4	2	2	0.77	0.29	**0.04**	0.75	0.25	**0.05**
lamport	●	1	2	0	0	4	2	2	0.88	0.31	**0.05**	0.88	0.29	**0.05**
peterson	●	1	3	0	0	4	2	2	0.66	0.26	**0.04**	0.65	0.25	**0.04**
szymanski	●	1	3	0	0	4	2	3	0.81	0.34	**0.07**	0.80	0.32	**0.04**
fib_longer_unsafe	●	6	2	0	0	10	6	2	**6.47**	8.19	94.84	6.51	**1.69**	135.45
fib_longer_safe		6	2	0	0	10	6	2	**9.78**	22.5	t.o.	**8.82**	31.8	t.o.
parker	●	1	2	0	0	4	2	3	1.68	0.31	**0.05**	2.19	0.28	**0.05**
stack_unsafe	●	2	2	1	2	5	2	2	1.50	0.41	**0.05**	1.49	0.35	**0.05**
litmus_safe (avg)		5	2	0	0	10	2	20	1.26	**0.17**	2.35	1.22	**0.15**	6.65
litmus_unsafe (avg)	●	5	2	0	0	10	2	20	1.27	**0.16**	3.86	1.26	**0.12**	1.58

timeout = 600s

faster than Nidhugg

Experimental Evaluation: Safestack

parameters			TSO analysis (3 bits)			CEX check (32 bits)		PSO analysis (3 bits)		CEX check (32 bits)	
K	N	rounds	Time	Mem.	Reach?	CEX?	Time	Time	Reach?	CEX?	Time
1	2	4	10m18s	0.8GB	Yes	Yes	23s	11m42s	Yes	Yes	4.82s
1	2	3	12m2s	0.6GB	No	-	-	11m16s	No	-	-
1	3	4	13m45s	1.2GB	Yes	Yes	30s	21m6s	Yes	Yes	6.40s
1	3	3	12m50s	0.9GB	No	-	-	12m20s	No	-	-
3	2	4	26m55s	1.4GB	Yes	Yes	24s	20m47s	Yes	Yes	4.33s
3	2	3	24m34s	1.0GB	No	-	-	27m15s	No	-	-
3	3	4	74m22s	3.4GB	Yes	Yes	31s	31m16s	Yes	Yes	5.47s
3	3	3	62m22s	1.0GB	Yes	Yes	30s	20m7s	Yes	Yes	2.84s
3	3	2	12m14s	0.6GB	No	-	-	11m14s	No	-	-
7	2	4	47m17s	2.4GB	Yes	Yes	27s	104m35s	Yes	Yes	6.05s
7	2	3	35m7s	1.3GB	No	-	-	36m14s	No	-	-

maxclock=K=1 forces SC analysis,
TSO puts 3x-4x overhead on lazy schema (SC times not shown in table)

Experimental Evaluation: Safestack

parameters			TSO analysis (3 bits)			CEX check (32 bits)		PSO analysis (3 bits)		CEX check (32 bits)	
K	N	rounds	Time	Mem.	Reach?	CEX?	Time	Time	Reach?	CEX?	Time
1	2	4	10m18s	0.8GB	Yes	Yes	23s	11m42s	Yes	Yes	4.82s
1	2	3	12m2s	0.6GB	No	-	-	11m16s	No	-	-
1	3	4	13m45s	1.2GB	Yes	Yes	30s	21m6s	Yes	Yes	6.40s
1	3	3	12m50s	0.9GB	No	-	-	12m20s	No	-	-
3	2	4	26m55s	1.4GB	Yes	Yes	24s	20m47s	Yes	Yes	4.33s
3	2	3	24m34s	1.0GB	No	-	-	27m15s	No	-	-
3	3	4	74m22s	3.4GB	Yes	Yes	31s	31m16s	Yes	Yes	5.47s
3	3	3	62m22s	1.0GB	Yes	Yes	30s	20m7s	Yes	Yes	2.84s
3	3	2	12m14s	0.6GB	No	-	-	11m14s	No	-	-
7	2	4	47m17s	2.4GB	Yes	Yes	27s	104m35s	Yes	Yes	6.05s
7	2	3	35m7s	1.3GB	No	-	-	36m14s	No	-	-

quicker to spot the bug under PSO
as it requires a smaller number of thread interactions;
performance comparable when no bugs are found

Experimental Evaluation: Safestack

parameters			TSO analysis (3 bits)			CEX check (32 bits)		PSO analysis (3 bits)		CEX check (32 bits)	
K	N	rounds	Time	Mem.	Reach?	CEX?	Time	Time	Reach?	CEX?	Time
1	2	4	10m18s	0.8GB	Yes	Yes	23s	11m42s	Yes	Yes	4.82s
1	2	3	12m2s	0.6GB	No	-	-	11m16s	No	-	-
1	3	4	13m45s	1.2GB	Yes	Yes	30s	21m6s	Yes	Yes	6.40s
1	3	3	12m50s	0.9GB	No	-	-	12m20s	No	-	-
3	2	4	26m55s	1.4GB	Yes	Yes	24s	20m47s	Yes	Yes	4.33s
3	2	3	24m34s	1.0GB	No	-	-	27m15s	No	-	-
3	3	4	74m22s	3.4GB	Yes	Yes	31s	31m16s	Yes	Yes	5.47s
3	3	3	62m22s	1.0GB	Yes	Yes	30s	20m7s	Yes	Yes	2.84s
3	3	2	12m14s	0.6GB	No	-	-	11m14s	No	-	-
7	2	4	47m17s	2.4GB	Yes	Yes	27s	104m35s	Yes	Yes	6.05s
7	2	3	35m7s	1.3GB	No	-	-	36m14s	No	-	-

increase maxlock to covers more reorderings,
more resource demanding..

Thank You

`users.ecs.soton.ac.uk/gp4/`**`cseq`**