Correct Parallel Software under Relaxed Memory Models

Berkeley Parlab

Jacob Burnim, Koushik Sen, Christos Stergiou

Jacob Burnim

Koushik Sen

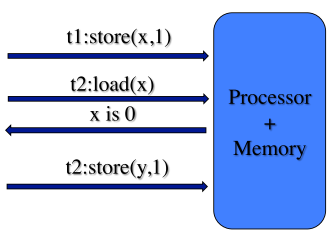
Christos Stergiou

Introduction

- Software developers write programs with intentional data races
  - Highly-concurrent libraries, lock-free data structures
  - Custom synchronization operations
  - Avoid cost of synchronization on certain frequent operations
- In the presence of data races, sequential consistency is no longer guaranteed
- Sequential Consistency (SC)
  - Lamport: *"… the result of any execution is the same as if the operations of all of the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.*
- Relaxed memory consistency:
  - Total Store Order (TSO): allows stores to be reordered past later loads, but maintains a total order over stores
  - Partial Store Order (PSO): TSO + allows stores to be reordered past later stores of different addresses
- Reasoning about memory models can be hard:

```
          Initially x = y = 0
   thread1:              thread2:
   1: x = 1               3: y=1
   2: t1 = y              4:  t2 = x
       assert(t1 == 1 || t2 == 1)
```

assertion can fail under TSO or PSO



Monitoring

Problem

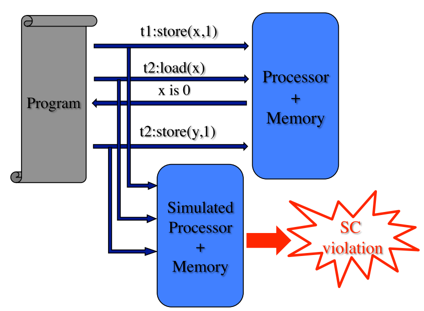
- Model-checking is intractable with added non-determinism from underlying memory-model

Idea

- Despite ah-hoc synchronization, programmers expect their program to be sequentially consistent
  - Sequential Consistency (SC) violations are likely to be bugs
- Can we find SC violations just by exploring SC executions of a program? [Burckhardt et al.]

Our Approach

- Devise monitoring algorithms for TSO and PSO
- Monitor algorithms are sound and complete
- Given SC violation, re-execute program and check if violation exposes a bug or not
- Based on intuitive operational simulation instead of complex axiomatic semantics
  - Yields simple algorithms (complex proofs)



	LOC	# SC schedules	TSO cycles	TSO bugs	PSO cycles	PSO bugs
dekker	23	220	3	2	5	2
bakery	31	1434	3	1	4	1
msn	83	616	0	-	3	3
ms2	78	500	0	-	2	1
lazylist	155	1764	0	-	2	1
harris	121	802	0	-	4	2
snark	150	1208	0	-	4	0

Active Testing

Problem

- Quickly find and reproduce memory model bugs
- Model checking can be expensive even with monitor
- Violations of sequential consistency are not always bugs

Our Solution: Active Testing

- 2-phase analysis and testing approach for predicting and confirming concurrency bugs
- Phase I: run program once and *predict* potential violations of sequential consistency
- Phase II: attempt to create potential violation by actively controlling thread schedule and underlying memory

```
              Initially x = y = done = 0
thread1 {                          thread2 {
1:   x = 1;                        4: if (done) {
2:   y = 1;                        5:   if (x==0)
3:   done = 1;                     6:     ERROR;
}                                  7:   local = y;
                                   }
```

Phase I predicts cycle : (1,3,4,5)

Phase II creates cycle by buffering write to x by thread1 and delaying thread2 at instruction 4

Bench mark	Cycles predicted	Cycles Confirmed			# of Bugs			Probability of confirming cycle		
		TSO	PSO	PSLO	TSO	PSO	PSLO	TSO	PSO	PSLO
dekker	112	47	45	69	39	38	65	0.69	0.81	0.84
bakery	222	36	75	100	33	68	96	0.85	0.84	0.82
msn	459	0	117	144	0	117	144	-	0.84	0.72
ms2	75	0	2	5	0	2	5	-	1.00	0.57
lazylist	192	0	8	10	0	8	9	-	0.96	0.62
harris	172	0	54	49	0	48	49	-	0.35	0.68
snark	1800	0	647	404	0	419	191	-	0.60	0.59