Predicting Null-Pointer Dereferences in Concurrent Programs

After work by: Azadeh Farzan
Parthasarathy Madhusudan
Niloofar Razavi
Francesco Sorrentino

Overview

- The problem
- The idea
- The solution
- The evaluation

The problem

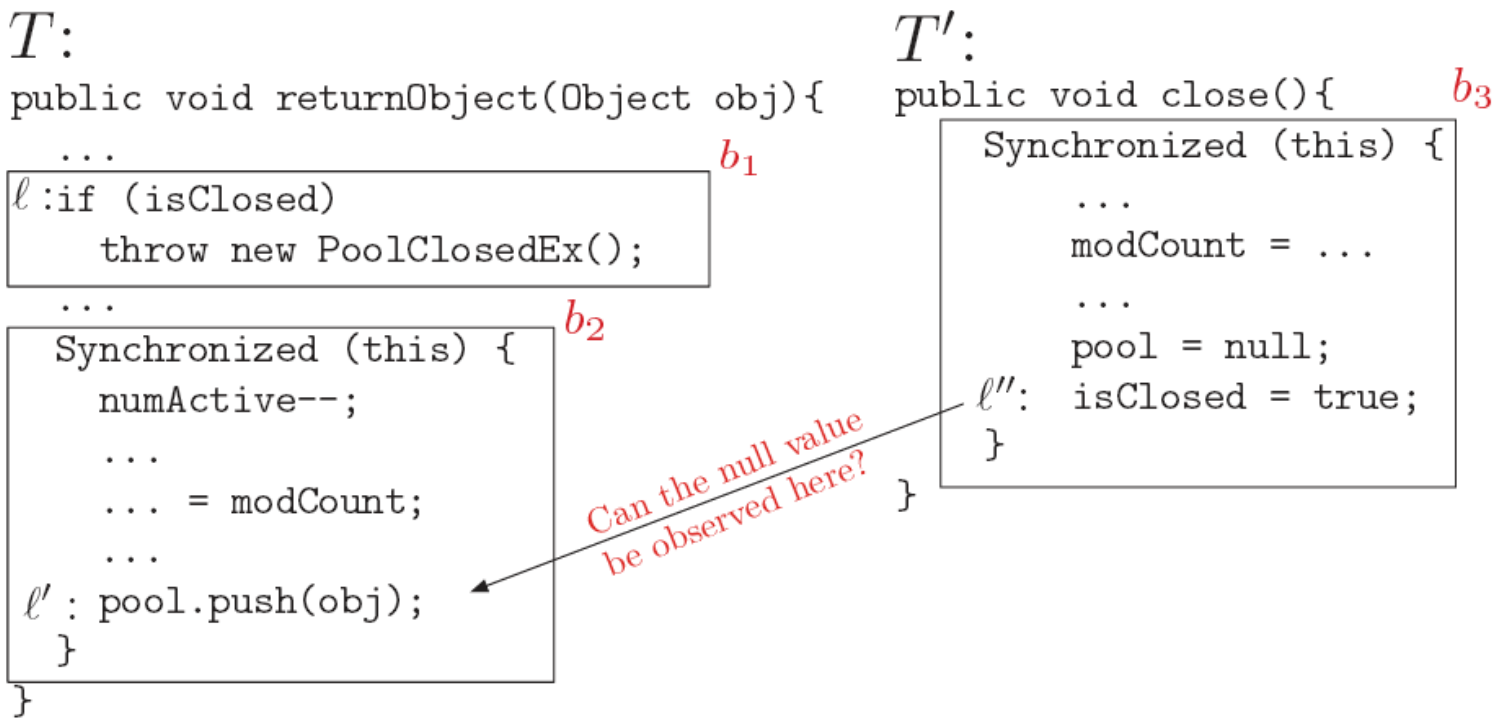
- Virtual memory segments can be assigned „at will” by the operating system.
- However, in many (in the case of C, arguably most) cases, the value that’s assigned to a pointer is not meant to be used as an offset to a memory location – usually it’s assigned there by mistake.
- To prevent invalid values being pulled from there, the OS prevents mapping the first (few) page(s) of the virtual memory space to physical memory address space.
- The MMU, via hardware traps, causes the OS to signal a software failure in case of an errorneous dereference.
- Most common consequence is having the process killed by the OS with a SIGSEGV signal (in case of POSIX systems).

But Java?

- No direct memory manipulation involved (viva la máquina virtual)...
- ...but the `null` value still has its uses:
  - Default reference-type pointer or pointer initialization in general; for example, we might want to keep said pointer in a clearly invalid state if it's yet to be assigned a valid object;
  - Signalling lack of further (valid) use for the object;
  - To speed up the GC taking care of the object by removing references (although it's non-deterministic and in general considered bad practice).

But Java?

- The problem is most prominent in the case of multi-threaded applications. A very simple example: a FIFO queue without an appropriate locking mechanism.
- A different example from the paper:



The idea

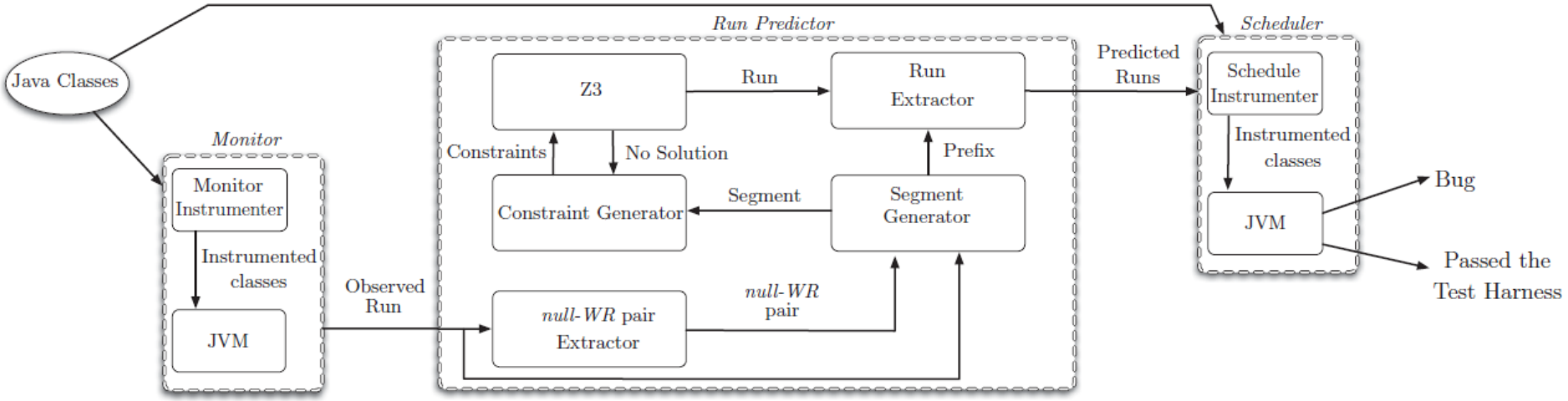
- Can’t quite avoid using `null` values in many cases;
- While Java offers a mechanism for graceful handling of null pointer dereferences (the `NullPointerExceptions`), performing manual checks is tedious and error-prone.
- What about the techniques for testing concurrent applications we already know?
  - A lot of them revolve around rescheduling operations done by different threads and simulating such runs in order to discover troublesome operation interleavings.

The idea

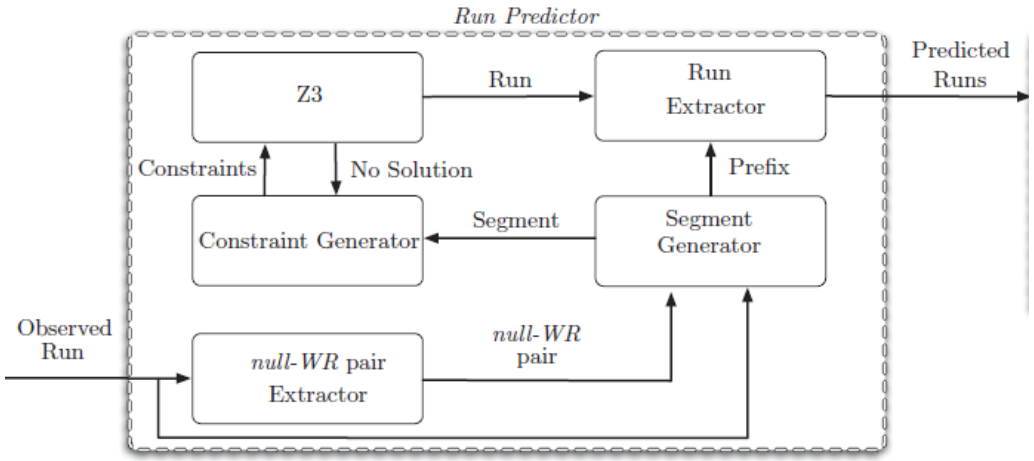
- What about the techniques for testing concurrent applications we already know?
  - A lot of them revolve around rescheduling operations done by different threads and simulating such runs in order to discover troublesome operation interleavings.
  - The goal would be to find such thread interleavings for which a write of `null` to a variable would be immediately (with regards to operations done on that variable) followed by a read, causing a `null` to be dereferenced.
  - Just permutating through possible interleavings won't do - we still need to account for scalability and reliability of the solution.

The solution

- Authors proposed the following framework for solving the problem of simulating runs causing a null pointer dereference:

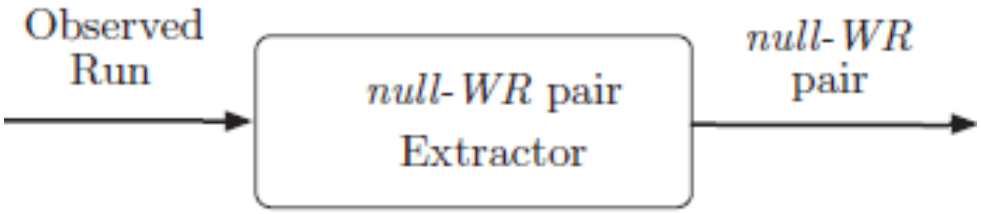


The solution



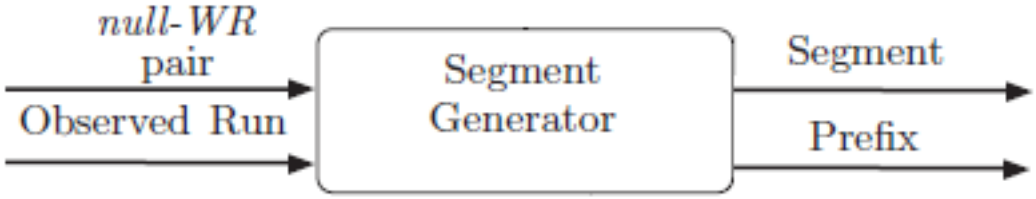
- The main entity the process revolves around is a *null-WR pair*.
- It's a pair α = (*e*, *f*), where *e* denotes an operation of assigning a `null` value to a reference and *f* denotes a read operation on that reference.
- The goal is to extract those pairs from a reference run and search for interleavings that will cause a `NullPointerException` to be raised.

The solution



- First, using static analysis, all *null-WR* pairs are detected;
- Then, for each of them, a lock validity test is performed to check if the pair is feasible in regards to locking semantics.
- All pairs that are left are then transferred to the segment generator.

The solution



- Every pair is analyzed in relation to the reference run in order to shorten the run to a segment including only relevant operations, i.e. those pertaining to the variable under test and to any locks that could potentially be used when the variable is used.

- The resulting segments are then passed to the SMT solver, while their prefixes left after prunning are passed to the run generator which will concatenate predicted feasible interleavings to it and pass it to the virtual machine for testing.

The solution

$$\psi \equiv PO \wedge CV \wedge DV \wedge LV$$

$$PO = (\bigwedge_{i=1}^{n} PO_i) \wedge C_{init}$$

$$C_{init} = \bigwedge_{i=1}^{n} (ts_{e_{init}} < ts_{e_{i,1}})$$

$$PO_i = \bigwedge_{j=1}^{m_i - 1} (ts_{e_{i,j}} < ts_{e_{i,(j+1)}})$$

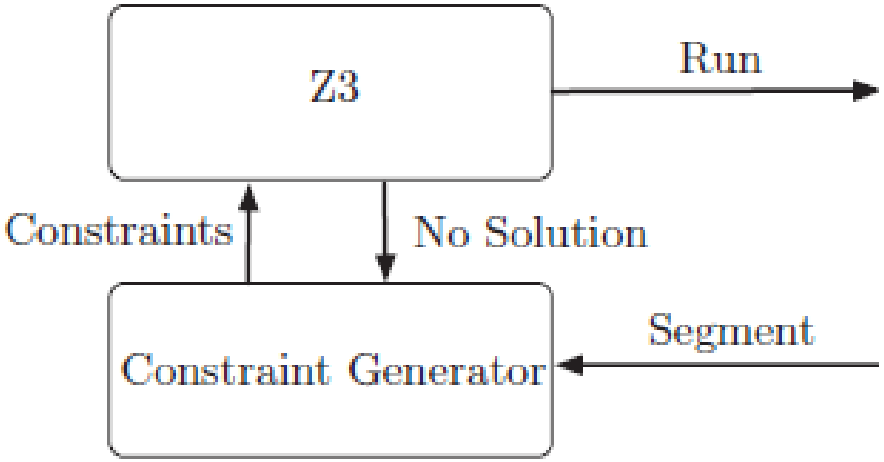
$$DV = \bigwedge_{x} \quad \bigwedge_{val \in Val(x)} \quad \bigwedge_{r \in R_{x,val}} \left(\bigvee_{w' \in W_{x,val}} Coupled_{r,w'} \right)$$

$$Coupled_{r,w} = (ts_w < ts_r) \wedge \bigwedge_{e'' \in W_x - \{w\}} ((ts_{e''} < ts_w) \vee (ts_r < ts_{e''}))$$

$$LV = LV_1 \wedge LV_2$$

$$LV_1 = \bigwedge_{i \neq j \in \{1,..,n\}} \bigwedge_{lock\ l} \bigwedge_{\substack{[e_{ac}, e_{rel}] \in L_{i,l} \\ [e'_{ac}, e'_{rel}] \in L_{j,l}}} \left(ts_{e_{rel}} < ts_{e'_{ac}} \vee\ ts_{e'_{rel}} < ts_{e_{ac}} \right)$$

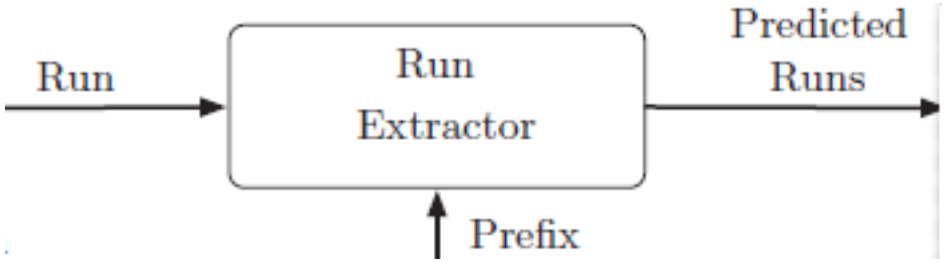
$$LV_2 = \bigwedge_{i \neq j \in \{1,..,n\}} \bigwedge_{lock\ l} \bigwedge_{\substack{e_{ac} \in NoRel_{i,l} \\ [e'_{ac}, e'_{rel}] \in L_{j,l}}} \left(ts_{e'_{rel}} < ts_{e_{ac}} \right)$$



- The segments are then analyzed in order to prepare a set of logical constraints in accordance to the maximal causal model, capturing sequential consistency, lock-, data- and thread creation validity.

- Such set of constraints is then passed to the Z3 SMT logical solver, which either passes the solution it found as series of event timestamps to the run generator or reports that no solution has been found; in such a case, data-validity constraints are iteratively relaxed until a solution is found.

The solution



- Predicted runs pertaining to the maximal causal model are then concatenated to the prefix taken from the segment generator and passed to the virtual machine's scheduler for testing.

- The framework also included a data-race detection unit which was not described in detail in the paper.

The evaluation

			Monitoring					Prediction			Scheduling				
Application (LOC)	Input	Base	Num. of Threads	Num. of Shared Variables	Num. of Locks	Num. of Potential Interleaving Points	Time to Monitor	Num. of *null-WR* Pairs	Num. of Precisely Predicted Runs	Additional Predicted Runs by Relaxation	Num. of Schedulable Predictions	Average Time per Predicted Run	Total Time	Null Pointer Deref. by Precise Prediction	Additional Null-Pointer Deref. by Relaxation
`Elevator` (566)	Data	7.3s	3	116	8	14K	7.4s	0	-	-	-	-	7.9s	**0**	**0**
	Data2	7.3s	5	168	8	30K	7.4s	0	-	-	-	-	8.9s	**0**	**0**
	Data3	19.2s	5	723	50	150K	19.0s	0	-	-	-	-	58.5s	**0**	**0**
`RayTracer` (1.5K)	A-10	5.0s	10	106	10	648	5.0s	9	9	-	9	5.6s	50.5s	**1***	**0**
	A-20	3.6s	20	196	20	1.7K	4.4s	19	19	-	19	6.7s	2m15s	**1***	**0**
	B-10	42.4s	10	106	10	648	42.5s	9	9	-	9	42.7s	6m24s	**1***	**0**
`Pool 1.2` (5.8K)	PT1	<1s	4	28	1	98	<1s	3	2	1	3	<1s	1.6s	**2**	**0**
	PT2	<1s	4	29	1	267	<1s	3	0	0	-	-	8.8s	**0**	**0**
	PT3	<1s	4	20	3	180	<1s	26	0	23	16	1.2s	27.0s	**0**	**3**
	PT4	<1s	4	24	3	360	<1s	32	2	21	15	2.5s	57.8s	**0**	**1**
`Pool 1.3` (7K)	PT1	<1s	4	30	1	100	<1s	3	0	3	3	<1s	2.6s	**0**	**0**
	PT2	<1s	4	31	1	271	<1s	3	0	0	-	-	9.8s	**0**	**0**
	PT3	<1s	4	20	3	204	<1s	35	0	30	19	1.4s	42.9s	**0**	**0**
	PT4	<1s	4	23	3	422	<1s	62	1	48	29	2.2s	1m49s	**0**	**1**
`Pool 1.5` (7.2K)	PT1	<1s	4	33	2	124	<1s	2	0	1	1	1.5s	1.5s	**0**	**0**
	PT2	<1s	4	34	2	306	<1s	5	0	1	0	10.5s	10.5s	**0**	**0**
	PT3	<1s	4	15	2	108	<1s	3	0	0	-	-	4.1s	**0**	**0**
	PT4	<1s	4	18	2	242	<1s	18	1	7	8	3.4s	27.4s	**0**	**1**
`SBucketMap` (750)	SMT	<1s	4	123	19	892	<1s	2	2	-	2	<1s	1.3s	**1**	**0**

The evaluation

			Monitoring					Prediction			Scheduling				
Application (LOC)	Input	Base	Num. of Threads	Num. of Shared Variables	Num. of Locks	Num. of Potential Interleaving Points	Time to Monitor	Num. of *null-WR* Pairs	Num. of Precisely Predicted Runs	Additional Predicted Runs by Relaxation	Num. of Schedulable Predictions	Average Time per Predicted Run	Total Time	Null Pointer Deref. by Precise Prediction	Additional Null-Pointer Deref. by Relaxation
Vector (1.3K)	VT1	<1s	4	44	2	370	<1s	21	11	10	21	<1s	14.3s	**2**	**0**
	VT2	<1s	4	34	2	536	<1s	31	21	10	31	1.1s	33.0s	**1**	**0**
	VT3	<1s	4	34	2	443	<1s	32	22	10	32	<1s	22.1s	**1**	**0**
	VT4	<1s	4	29	2	517	<1s	30	0	30	30	2s	59.4s	**0**	**1***
	VT5	<1s	4	29	2	505	<1s	85	1	84	82	2s	2m57s	**0**	**1***
Stack (1.4K)	ST1	<1s	4	29	2	205	<1s	11	6	5	11	<1s	5.5s	**2**	**0**
	ST2	<1s	4	24	2	251	<1s	16	11	5	15	<1s	10.9s	**1**	**0**
	ST3	<1s	4	24	2	248	<1s	17	12	5	17	<1s	10.3s	**1**	**0**
	ST4	<1s	4	29	2	515	<1s	30	0	30	30	1.8s	53.2s	**0**	**1***
	ST5	<1s	4	29	2	509	<1s	85	1	84	83	2.0s	2m51s	**0**	**1***
HashSet (1.3K)	HT1	<1s	4	76	1	432	<1s	7	7	-	7	<1s	3.2s	**1**	**0**
	HT2	<1s	4	54	1	295	<1s	0	-	-	-	-	<1s	**0**	**0**
StringBuffer (1.4K)	SBT	<1s	3	16	3	80	<1s	2	2	-	2	<1s	1.3s	**1**$^{+}$	**0**
Apache FtpServer (22K)	LGN	1m2s	4	112	4	582	60s	116	78	32	65	1m13s	2h14m46s	**9**	**3**
Hedc (30K)	Std	1.7s	7	110	6	602	1.74s	18	9	1	10	11.7s	1m57s	**1**	**0**
Weblech v.0.0.3 (35K)	Std	4.9s	3	153	3	1.6K	4.92s	55	10	29	30	16.26s	10m34s	**1**	**1**$^{@}$
											Total Number of Errors			**27**	**14**

The evaluation

- The prediction and scheduling time is fairly reasonable, considering the use of a sophisticated logic solver.
- The results of benchmarks turned out to be very good: about 40 exceptions with null-pointer dereferences and 60 data races, which „is the most successful attempt at finding errors on these benchmarks in the literature”.
- All errors were also found to be deterministically reproducible using the scheduler.

Thank you for your attention.

Any questions?