

TESTING CONCURRENT DATA STRUCTURES: LINCHECK

Sofia Gavanelli

DIFFICOLTÀ NELLA PROGRAMMAZIONE CONCORRENTE

Esempi di problematiche che si possono incontrare:

- Gestione dell'accesso alla memoria condivisa
- **Deadlock:** due thread in un'attesa infinita del rilascio di una risorsa senza rilasciare il lock già ottenuto
- **Starvation:** quando un processo è impossibilitato a ottenere le risorse che gli servono anche se queste sono disponibili
- **Livelock:** entrambi i thread rilasciano il loro lock a un certo punto ma senza ottenere tutte le risorse necessarie
- **Race condition**

LINCHECK

Approccio dichiarativo: segnalare le operazioni della struttura dati

⇒ Non *come* testare la struttura dati ma *cosa* testare

⇒ Generazione automatica di scenari concorrenti

⇒ Stress testing o model checking

⇒ Condizione di correttezza: **linearizzabilità**

“Un calcolo concorrente è linearizzabile se è equivalente a un calcolo sequenziale legale.”

ESEMPIO CONCURRENTLINKEDQUEUE

```
1 class DequeTest {
2   val deque = ConcurrentLinkedDeque<Int>()
3
4   @Operation fun addFirst(e: Int) = deque.addFirst(e)
5   @Operation fun addLast(e: Int)  = deque.addLast(e)
6   @Operation fun pollFirst()      = deque.pollFirst()
7   @Operation fun pollLast()       = deque.pollLast()
8   @Operation fun peekFirst()      = deque.peekFirst()
9   @Operation fun peekLast()       = deque.peekLast()
10
11   @Test fun runTest() = ModelCheckingOptions()
12                        .check(this::class)
13 }
```

RISULTATI CONCURRENTLINKEDQUEUE

```
= Invalid execution results =  
| addLast(-6)      | addFirst(-8)    |  
| peekFirst(): -8 | pollLast(): -8 |
```

= The following interleaving leads to the error =

```
|      | addFirst(-8)      |  
|      | pollLast()        |  
|      | pollLast(): -8 at DequeTest.pollLast(DequeTest.kt:35) |  
|      | last(): Node@1 at CLD.pollLast(CLD.java:936) |  
|      | item.READ: null at CLD.pollLast(CLD.java:938) |  
|      | prev.READ: Node@2 at CLD.pollLast(CLD.java:946) |  
|      | item.READ: -8 at CLD.pollLast(CLD.java:938) |  
|      | next.READ: null at CLD.pollLast(CLD.java:940) |  
|      | switch            |  
| addLast(-6)      |  
| peekFirst(): -8  |  
|      | item.CAS(-8,null): true at CLD.pollLast(CLD.java:941) |  
|      | unlink(Node@2) at CLD.pollLast(CLD.java:942) |  
|      | result: -8        |
```

PROPRIETÀ

- Testing dichiarativo
- Nessuna restrizione di implementazione
- No falsi positivi (report di soli errori riproducibili)
- User-friendly (tracce degli errori)
- Flessibilità (custom scenario generation)

OVERVIEW

- Lavora in fasi:
 1. **Generazione** degli scenari
 2. **Esecuzione** degli scenari
 3. **Verifica** dei risultati
- Minimizing failing scenarios: non ottimale teoricamente, ma praticamente funzionante

FASE 1: GENERAZIONE DEGLI SCENARI

- Possibilità di gestire il numero dei thread paralleli, delle operazioni al loro interno e degli scenari generati:

iterations - n of different scenarios

invocationsPerIteration - n of invocations for each scenario

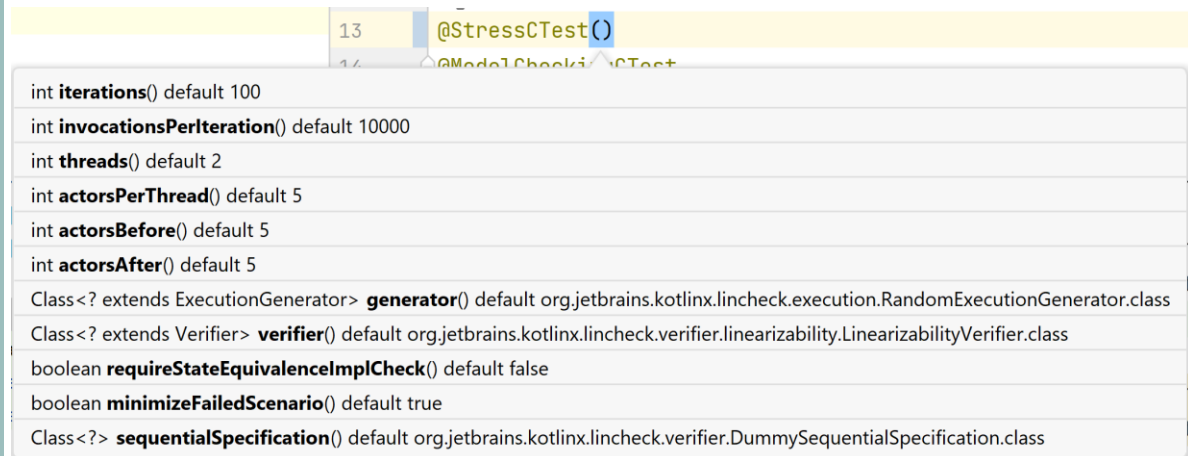
threads - n of threads

actorsPerThread - n of op. executed in each thread

actorsBefore - n of op. to be executed before the concurrent part, sets up a random initial state

(**actorsAfter** - number of op. after the concurrent part, helps to verify that a data structure is still correct)

verifier - verifier for an expected correctness contract



```
13 @StressCTest()
14 @ModelCheck(...)
...
int iterations() default 100
int invocationsPerIteration() default 10000
int threads() default 2
int actorsPerThread() default 5
int actorsBefore() default 5
int actorsAfter() default 5
Class<? extends ExecutionGenerator> generator() default org.jetbrains.kotlinx.lincheck.execution.RandomExecutionGenerator.class
Class<? extends Verifier> verifier() default org.jetbrains.kotlinx.lincheck.verifier.linearizability.LinearizabilityVerifier.class
boolean requireStateEquivalenceImplCheck() default false
boolean minimizeFailedScenario() default true
Class<?> sequentialSpecification() default org.jetbrains.kotlinx.lincheck.verifier.DummySequentialSpecification.class
```


GENERAZIONE DEGLI SCENARI

- Possibile restrizione dei valori degli input:

```
@Param(name = "key", gen = IntGen.class, conf = "1:7")
@StressCTest
@ModelCheckingCTest
public class HashMapLinearizabilityTest {
    private HashMap<Integer, Integer> map = new HashMap<>();

    @Operation
    public Integer put(@Param(name = "key") int key, int value) { return map.put(key, value); }
```

- Possibilità di avere operazioni non eseguite in concorrenza

FASE 2: ESECUZIONE DEGLI SCENARI

Lincheck utilizza stress testing e model checking per esaminare gli scenari

- **Stress testing:** lo scenario è eseguito su thread paralleli numerose volte per individuare degli incroci (di operazioni) che producono risultati errati
- **Model checking:** Lincheck esamina diversi incroci con un numero limitato di context switch

Il model checking aumenta il test coverage e chiarifica il trace di esecuzione dei comportamenti errati. **Tuttavia**, questa implementazione presuppone un modello di memoria sequentially consistent che può risultare in una perdita dei bug dovuti a effetti low-level (come può essere un volatile dimenticato).

ESECUZIONE DEGLI SCENARI

Lincheck **studia** tutti gli **incroci** (interleavings) con un solo context switch ma lo fa **in maniera equa**, cercando di esplorare una varietà di incroci contemporaneamente. Quando tutti gli incroci con un solo context switch sono conclusi allora Lincheck passa a quelli con due context switch e **ripete** il procedimento **aumentando** ogni volta il numero di **switch**.

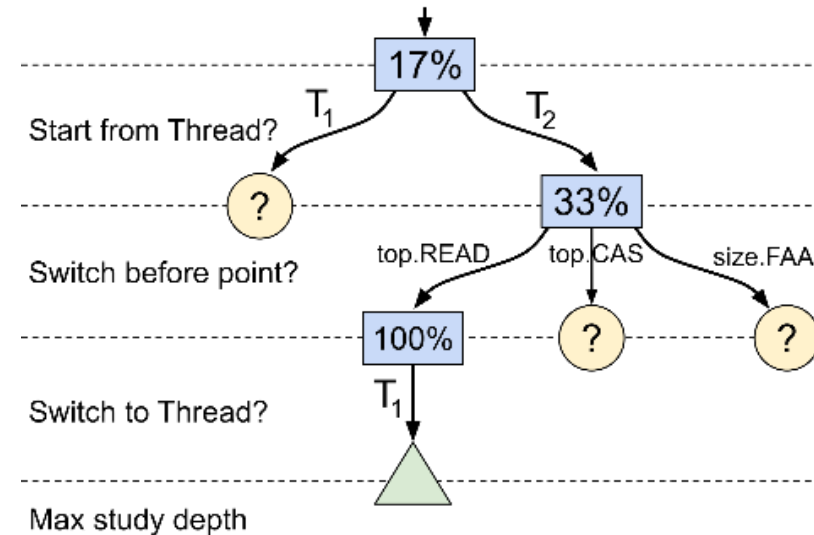
Questa strategia **aumenta il coverage** e permette di ottenere lo schedule errato con il numero minore di context switch.

- **Switch Points:** Lincheck inserisce switch point nel codice per controllarne l'esecuzione. Questi **punti** identificano dove può essere effettuato un **context switch**. Per inserire uno switch point il codice da testare è trasformato tramite il framework ASM per aggiungere le invocazioni alle funzioni interne prima di un accesso. Questa trasformazione è effettuata sul momento utilizzando un custom class loader.

ESECUZIONE DEGLI SCENARI

- **Interleaving Tree:** Lincheck, per esplorare i possibili schedule, costruisce un interleaving tree dove gli archi rappresentano le diverse decisioni che possono essere prese dallo scheduler.

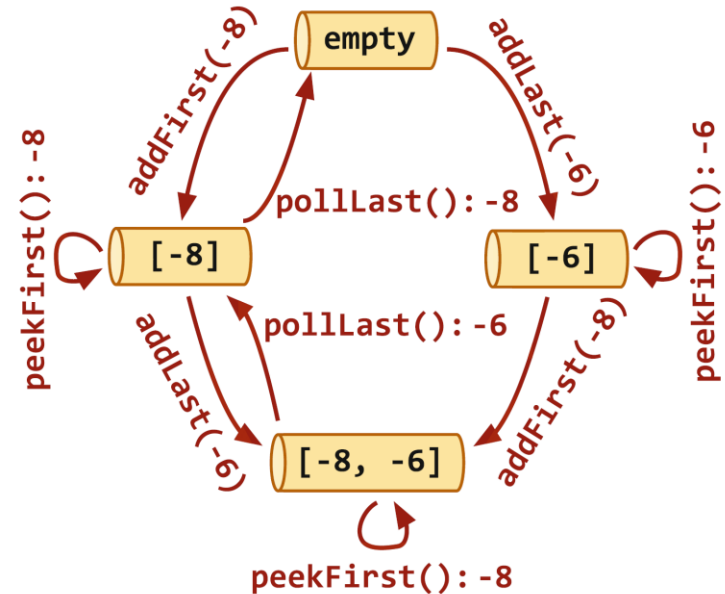
```
= Invalid execution results =  
| push(7): void | pop(): 7 |  
|               | size(): -1 |  
  
= The following interleaving leads to the error =  
| push(7)  
|   top.READ: null  
|   at Stack.push(Stack.kt:5)  
|   top.compareAndSet(null,Node@1): true  
|   at Stack.push(Stack.kt:7)  
|   switch  
|  
|   pop(): 7  
|   size(): -1  
|   thread is finished  
|  
|   _size.incrementAndGet(): 0  
|   at Stack.push(Stack.kt:8)  
|   result: void  
|   thread is finished
```



FASE 3: VERIFICA DEI RISULTATI OTTENUTI

- Proprietà di correttezza: **linearizability** (default), quiescent consistency, quantitative relaxation e quasi-linearizability
- Utilizzo di un LTS (labeled transition system):
Se si trova un cammino finito sul grafo che faccia ottenere gli stessi risultati generati dagli scenari allora i risultati sono considerati validi

```
= Invalid execution results =  
| addLast(-6)      | addFirst(-8)  |  
| peekFirst(): -8 | pollLast(): -8 |
```



VALUTAZIONE

- Numerosi nuovi bug scoperti
- Le differenti configurazioni sono pensate per local builds e CI servers:
 - **Fast:** 30 scenari di 2 thread × 3 operazioni, 1000 invocazioni per ognuno
 - **Long:** 100 scenari di 3 thread × 4 operazioni, 10000 invocazioni per ognuno

Data Structure	Fast Configuration		Long Configuration	
	Stress	MC	Stress	MC
ConcurrentHashMap (Java)	0.3 s	2.7 s	38.1 s	1 m 44 s
ConcurrentLinkedQueue (Java)	0.4 s	1.7 s	1 m 26 s	1 m 41 s
LockFreeTaskQueue (Kotlin Coroutines)	1.1 s	1.4 s	39.6 s	54.8 s
Semaphore (Kotlin Coroutines)	2.1 s	3.6 s	22.3 s	1 m 44 s
ConcurrentLinkedDeque (Java)	0.4 s	1.2 s	19.7 s	10.7 s
AbstractQueueSynchronizer (Java)	1.6 s	0.5 s	18.2 s	8.6 s
Mutex(Kotlin Coroutines)	0.9 s	2.6 s	23.6 s	8.7 s
NonBlockingHashMapLong (JCTools)	0.6 s	1.3 s	4.4 s	7 s
ConcurrentRadixTree ([29])	2.9 s	10.6 s	40.9 s	2 m 30 s
SnapTree [31]	1.7 s	5.8 s	38.4 s	5 m 6 s
LogicalOrderingAVL [32]	1.5 s	4.2 s	17.1 s	36.9 s
CATree [33]	20.1 s	0.8 s	41.3 s	6.5 s
ConcurrencyOptimalTree [34]	0.4 s	1.5 s	3 s	7.3 s

RIFERIMENTI

- Koval, N., Fedorov, A., Sokolova, M., Tsitelov, D., Alistarh, D. (2023). **Lincheck: A Practical Framework for Testing Concurrent Data Structures on JVM**. In: Enea, C., Lal, A. (eds) Computer Aided Verification. CAV 2023. Lecture Notes in Computer Science, vol 13964. Springer, Cham. https://link.springer.com/content/pdf/10.1007/978-3-031-37706-8_8.pdf?pdf=inline%20link
- <https://blog.jetbrains.com/kotlin/2021/02/how-we-test-concurrent-primitives-in-kotlin-coroutines/>
- <https://github.com/devexperts/lin-check/tree/master>
- Herlihy, M.P., Wing, J.M.: **Linearizability: a correctness condition for concurrent objects**. ACM Trans. Program. Lang. Syst. (TOPLAS) 12(3), 463–492 (1990)
<https://cs.brown.edu/~mph/HerlihyW90/p463-herlihy.pdf>

