Asynchronous Assertions

by Edward E. Afandilian, Samuel Z. Guyer, Martin Vechev and Eran Yahav, OOPSLA'11

presented by David Herzog-Botzenhart as part of the seminar "Concurrency and Memory Management" supervised by Professor Christoph Kirsch, April 2013

Assertions

- Assertions - A powerful and convenient tool.
- The problem with Synchronous Assertion.
- The idea of Asynchronous Assertion (STROBE).
- Two ways of dealing with a failed Assertion

Snapshotting

- Snapshot semantics.
- Checker threads.

Experimental Evaluation

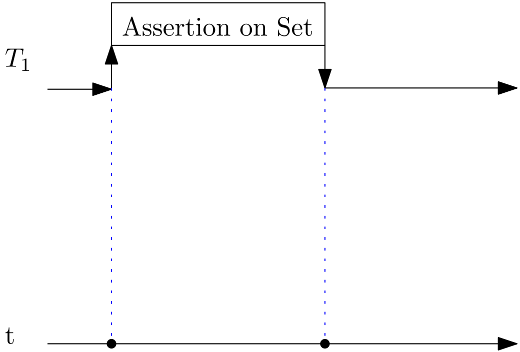
- Experimental Evaluation

Synchronous Assertion at runtime.

```
// A linked list implementation of a set data structure
    .
public class Set {
...
    public void addElement(Object newElement){...}
}

 ... // somewhere at initialization.
Set setOfArrivingAirplanes = new Set();
 ... // somewhere at runtime.
setOfArrivingAirplanes.add(newElement);
 ... //critical check (never miss an arriving plane ;)
Assert.true(setOfArrivingAirplanes.contains(newElement)
    );
 ...
```

Synchronous Assertion at runtime.



Assertions ... an example

Assertion of invariants of a Concurrent Sorted Set Collection based on a sorted list data structure.
The Set's invariants are:

- No two equal objects may be in there.
- Insertion of an other equal object fails.
  (will return the equal object).
- The Collection is sorted (at any time).

Requirement: Program has to terminate if invariant is violated.

Assertions ... an example

Assertion of invariants of a Concurrent Sorted Set Collection based on a sorted list data structure.
The Set's invariants are:

- No two equal objects may be in there.
- Insertion of an other equal object fails.
  (will return the equal object).
- The Collection is sorted (at any time).

Requirement: Program has to terminate if invariant is violated.

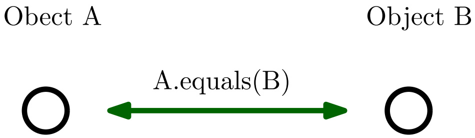
Assertions ... an example

Assertion of invariants of a Concurrent Sorted Set Collection based on a sorted list data structure.
The Set's invariants are:

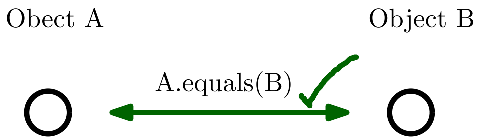
- No two equal objects may be in there.
- Insertion of an other equal object fails.
  (will return the equal object).
- The Collection is sorted (at any time).

Requirement: Program has to terminate if invariant is violated.

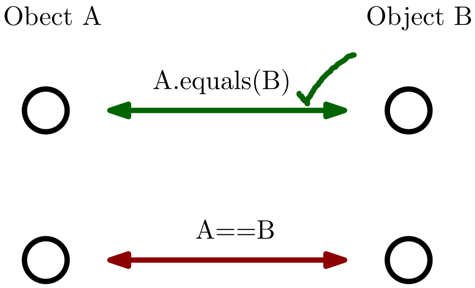
Objects with same order.



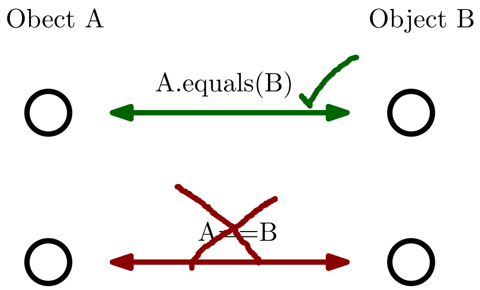
Objects with same order.



Objects with same order.

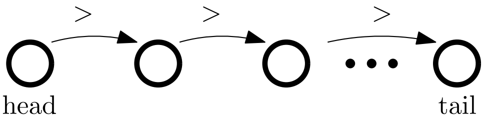


Objects with same order.

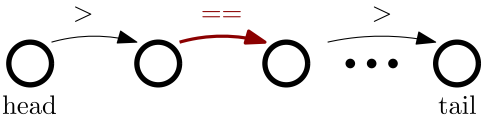


Set as sorted list

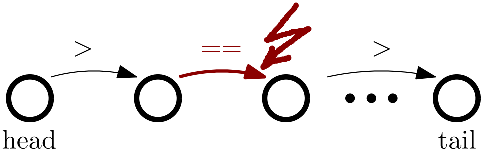
Set as sorted list - The invariant.



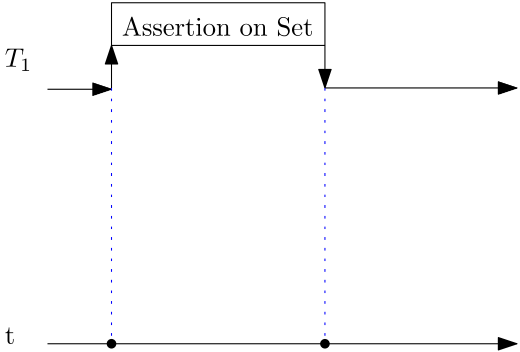
Set as sorted list - The violation of the invariant.



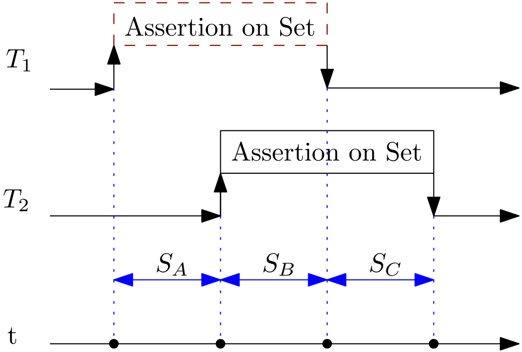
Set as sorted list - The violation of the invariant.



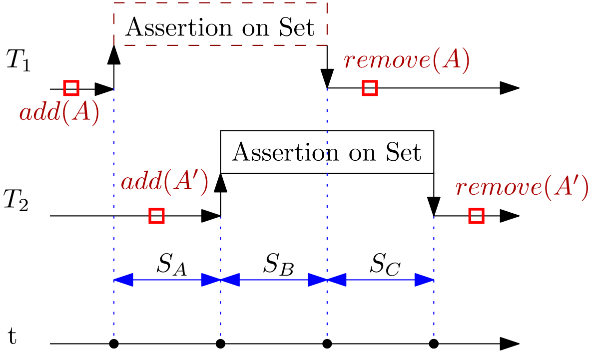
Synchronous Assertion in single thread, to expensive?



How can such an Assertion of the invariant fail?



How can such an Assertion of the invariant fail?



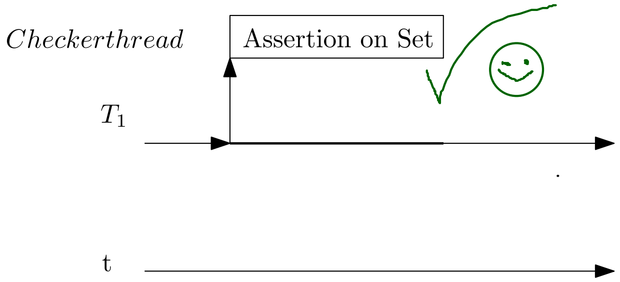
Synchronous Assertions - Problems?

- Cost of inline assertions at runtime.
- Concurrency issues on shared data structures.

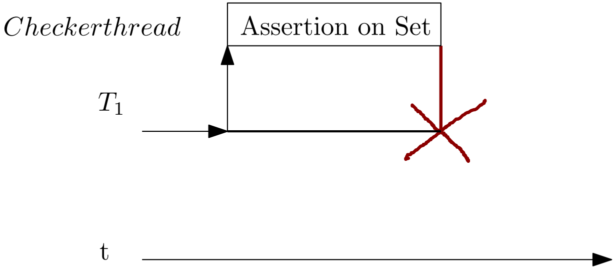
Synchronous Assertions - Problems?

- Cost of inline assertions at runtime.
- Concurrency issues on shared data structures.

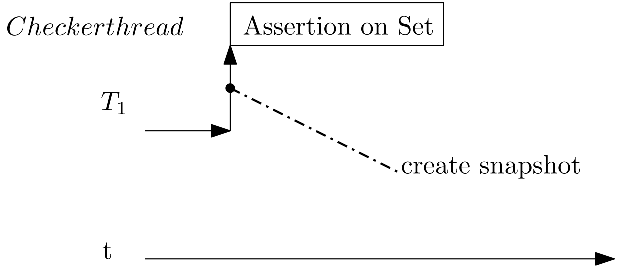
Asynchronous Assertion - Assertion thread(s)



Asynchronous Assertion - Assertion thread(s)



Asynchronous Assertion - Assertion thread(s)



Two way, no exit?

The program continues executing!

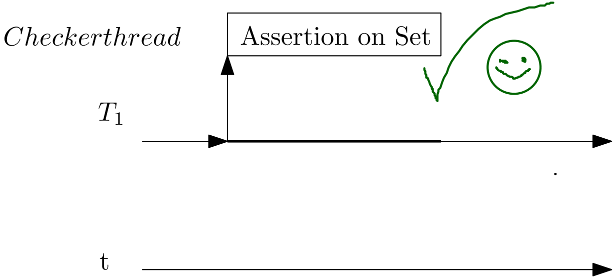
- The checker thread runs and terminates (traditionally). (Strobe)
- The program must not progress beyond a point. (FutureStrobe)

Two way, no exit?

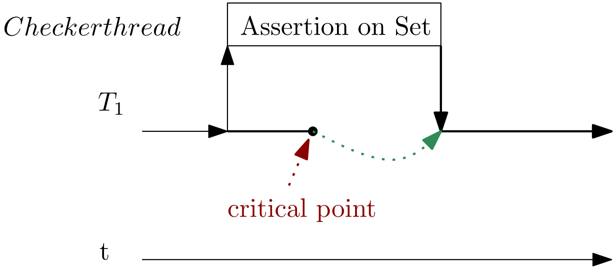
The program continues executing!

- The checker thread runs and terminates (traditionally). (Strobe)
- The program must not progress beyond a point. (FutureStrobe)

The **Strobe** Assertion.



The **FutureStrobe** Assertion.



Snapshotting Guarantees

It is guaranteed that either:

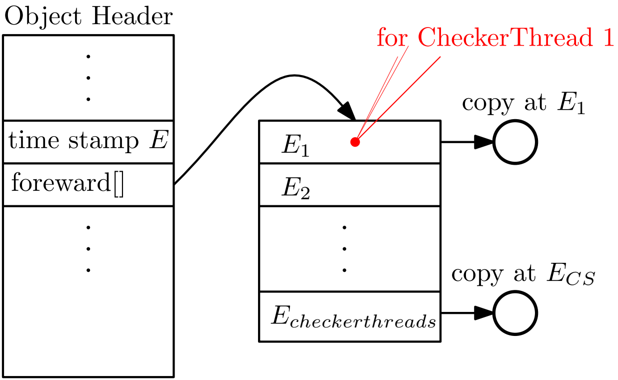
- We see the original object, if it has not been modified.
- A copy of the object that reflects the state at the time the Assertion started.

Snapshotting Guarantees

It is guaranteed that either:

- We see the original object, if it has not been modified.
- A copy of the object that reflects the state at the time the Assertion started.

Object Header



Creation of a Snapshot, the epoch counter

- The **epoch counter** E increments each time an Assertion is triggered (initially 0).
- Each Assertion is a done in a **Checker Thread** that owns its own Snapshot.
- "Object write" **creates a copy** of the object with current E as unique identifier (timestamp) before write.
- Each Assertion is a done in a **Checker Thread** out of the Checker Thread pool.
- The number of Checker Threads is constant (limited).

Creation of a Snapshot, the epoch counter

- The **epoch counter** E increments each time an Assertion is triggered (initially 0).
- Each Assertion is a done in a **Checker Thread** that owns its own Snapshot.
- "Object write" **creates a copy** of the object with current E as unique identifier (timestamp) before write.
- Each Assertion is a done in a **Checker Thread** out of the Checker Thread pool.
- The number of Checker Threads is constant (limited).

Creation of a Snapshot, the epoch counter

- The **epoch counter** E increments each time an Assertion is triggered (initially 0).
- Each Assertion is a done in a **Checker Thread** that owns its own Snapshot.
- "Object write" **creates a copy** of the object with current E as unique identifier (timestamp) before write.
- Each Assertion is a done in a **Checker Thread** out of the Checker Thread pool.
- The number of Checker Threads is constant (limited).

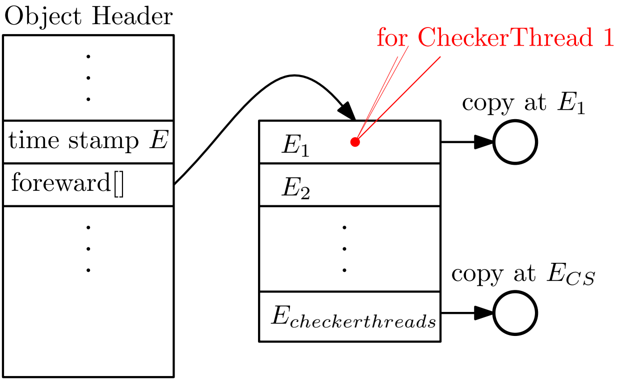
Creation of a Snapshot, the epoch counter

- The **epoch counter** E increments each time an Assertion is triggered (initially 0).
- Each Assertion is a done in a **Checker Thread** that owns its own Snapshot.
- "Object write" **creates a copy** of the object with current E as unique identifier (timestamp) before write.
- Each Assertion is a done in a **Checker Thread** out of the Checker Thread pool.
- The number of Checker Threads is constant (limited).

Creation of a Snapshot, the epoch counter

- The **epoch counter** E increments each time an Assertion is triggered (initially 0).
- Each Assertion is a done in a **Checker Thread** that owns its own Snapshot.
- "Object write" **creates a copy** of the object with current E as unique identifier (timestamp) before write.
- Each Assertion is a done in a **Checker Thread** out of the Checker Thread pool.
- The number of Checker Threads is constant (limited).

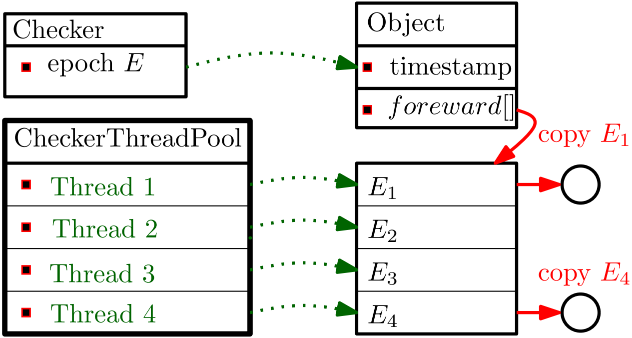
Object Header



The **modifiedAt(Object o)** method.

The **modifiedAt(o)** method returns the epoch (timestamp) in which the object was last modified.
An object needs to be preserved for an assertion started at E_t if it was modified in an epoch **before** E_t

Overview of Snapshots and copied (snapshot) Objects



Synchronization

- The operation *modifiedAt(o)* is synchronized.
- Using two techniques to avoid race conditions by updating the timestamp.
  - A sentinel value ensures that the three operations are atomic.
    - Copying the object.
    - Updating the epoch counter E.
    - Performing the write.
  - Ordering the operations in such a way that the checker thread cannot see intermediate results.

Synchronization

- The operation *modifiedAt(o)* is synchronized.
- Using two techniques to avoid race conditions by updating the timestamp.
  - A sentinel value ensures that the three operations are atomic.
    - Copying the object.
    - Updating the epoch counter E.
    - Performing the write.
  - Ordering the operations in such a way that the checker thread cannot see intermediate results.

Synchronization

- The operation *modifiedAt(o)* is synchronized.
- Using two techniques to avoid race conditions by updating the timestamp.
  - A sentinel value ensures that the three operations are atomic.
    - Copying the object.
    - Updating the epoch counter E.
    - Performing the write.
  - Ordering the operations in such a way that the checker thread cannot see intermediate results.

Synchronization

- The operation *modifiedAt(o)* is synchronized.
- Using two techniques to avoid race conditions by updating the timestamp.
  - A sentinel value ensures that the three operations are atomic.
    - Copying the object.
    - Updating the epoch counter E.
    - Performing the write.
  - Ordering the operations in such a way that the checker thread cannot see intermediate results.

The write barrier implementation.

The write barrier is synchronized operation with a **BEING_COPIED** sentinel (semaphore) for the timestamp.

- If the timestamp is current, no copy is needed.
- If the timestamp is older, a copy is created using the sentinel, to prevent concurrent copies.

The write barrier implementation.

The write barrier is synchronized operation with a **BEING_COPIED** sentinel (semaphore) for the timestamp.

- If the timestamp is current, no copy is needed.
- If the timestamp is older, a copy is created using the sentinel, to prevent concurrent copies.

Snapshot semantics.

```
// -- Returns either the original or the copy
Object readBarrier(Object obj) {
  // -- Get forwarding array
  Object[] forwardArr = Header.getForwardingArray(obj)
  // -- No forwarding array? return original
  if (forwardArr == null) {
    return obj; }
  else {
    // -- Else load copy from forwarding array,
    // indexing by checking thread ID
    Object copy = forwardArray[thisThread.checkerId];
    // -- No copy of this object? return original
    if(copy == null){return obj;}
    // -- ...otherwise return copy (snapshot)
    else { return copy; }
  }
}
```

Checker Threads?

The number of synchronous Assertions is limited to the number of CheckerThreads (Checker Thread pool).
If the maximum number of asynchronous checker threads is reached, the new Assertion is blocked.

Experimental Evaluation

Here are only some citations of the experimental results. Of course the setups vary and therefore the following findings are not to be accounted as generally true.

- The main source of overhead is creating and maintaining snapshots.
- About $\frac{1}{3}$ of overhead is due to other factors such as cost of extra words in the header, the epoch and possibly additional pressure for the system.
- On average, GC time increased by 10% to 50%. But GC time accounted only 5% to 10%, so the impact was low.

Experimental Evaluation

Here are only some citations of the experimental results. Of course the setups vary and therefore the following findings are not to be accounted as generally true.

- The main source of overhead is creating and maintaining snapshots.
- About $\frac{1}{3}$ of overhead is due to other factors such as cost of extra words in the header, the epoch and possibly additional pressure for the system.
- On average, GC time increased by 10% to 50%. But GC time accounted only 5% to 10%, so the impact was low.

Experimental Evaluation

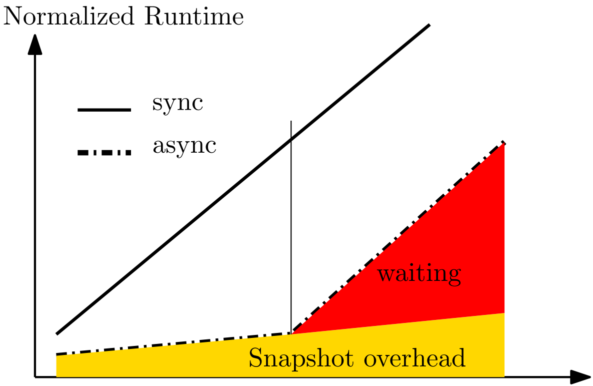
Here are only some citations of the experimental results. Of course the setups vary and therefore the following findings are not to be accounted as generally true.

- The main source of overhead is creating and maintaining snapshots.
- About $\frac{1}{3}$ of overhead is due to other factors such as cost of extra words in the header, the epoch and possibly additional pressure for the system.
- On average, GC time increased by 10% to 50%. But GC time accounted only 5% to 10%, so the impact was low.

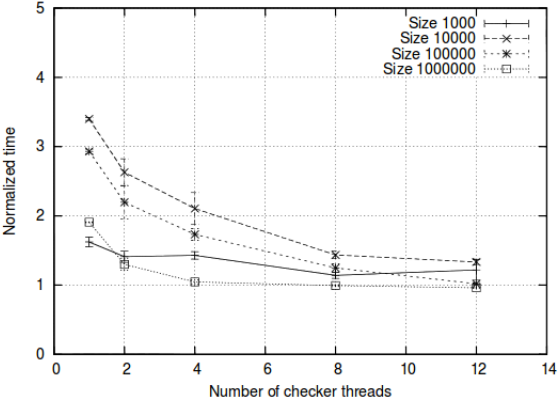
Experimental Evaluation

- When there are not enough checker threads, the wait for becomes significant. The slowdown grows similar to the synchronous Assertion.

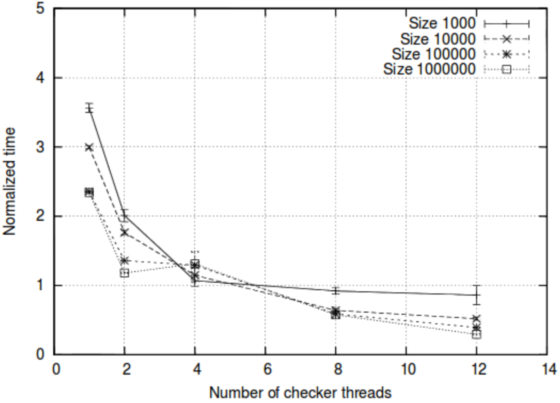
Assertions workload overwhelms Checker threads



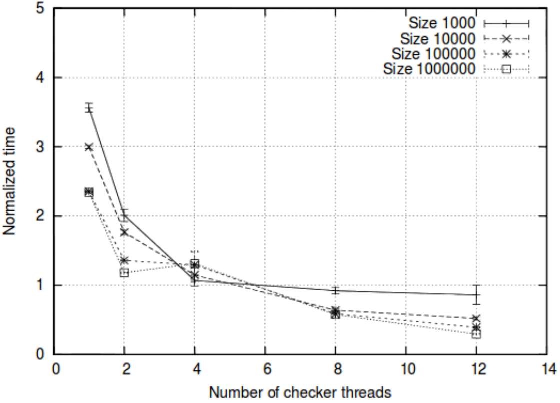
Overhead versus number of checker threads. (linked list)



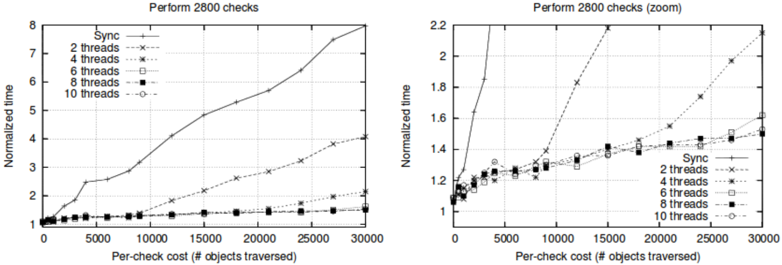
Overhead versus number of checker threads. (tree map)



Overhead versus number of checker threads. (tree map)



Overhead versus number of checker threads. (tree map)



Overhead of a fixed number (2800) of assertions for a range of costs, normalized to baseline (no assertions, unmodified RVM).

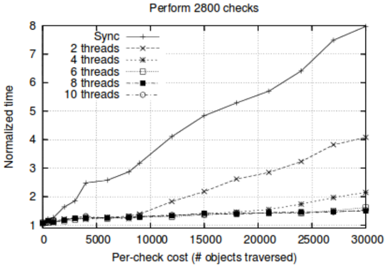
Fixed cost, vary frequency.

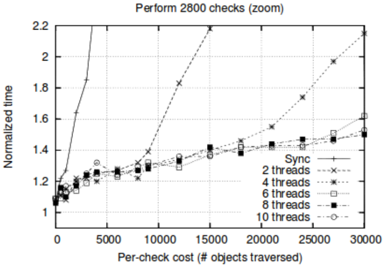
- The overhead of synchronous checking grows very rapidly as assertions become more frequent.
- The overhead of asynchronous checking grows slowly as long as the checker threads can keep up with the rate of the assertions.

Fixed cost, vary frequency.

- The overhead of synchronous checking grows very rapidly as assertions become more frequent.
- The overhead of asynchronous checking grows slowly as long as the checker threads can keep up with the rate of the assertions.

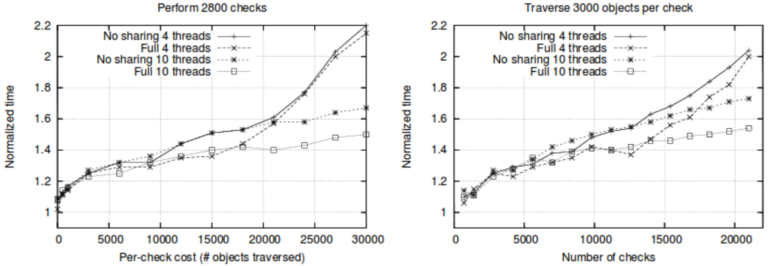
Overhead of fixed-cost assertions, over a range of assertion frequencies





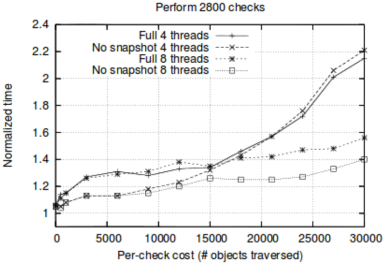
Traverse 6000 objects, no assertions, unmodified.

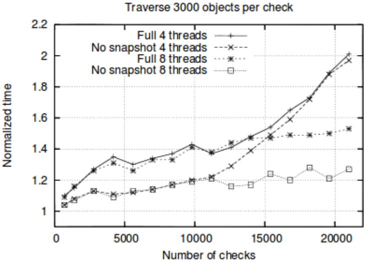
Effect of not sharing objects.



Overhead increases by 25% to 30% over unmodified STROBE implementation.

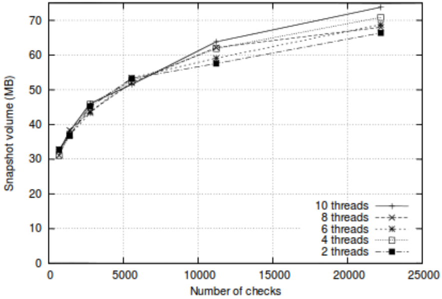
Evaluate asynchronous assertions without creating snapshots.





Creating and maintaining snapshots accounts approximately $\frac{2}{3}$ of the overhead.

Copying costs and GC time.



More checks require more snapshots, increasing the volume of bytes copied, but tops out as all heap writes generate copies.

This is the end... [The Doors]