Asynchronous Assertions

Edward E. Aftandilian
Tufts University and Google
eaftan@cs.tufts.edu

Samuel Z. Guyer
Tufts University
sguyer@cs.tufts.edu

Martin Vechev
ETH Zurich and IBM Research
martin.vechev@gmail.com

Eran Yahav [*]
Technion, Israel
yahave@cs.technion.ac.il

Abstract

Assertions are a familiar and widely used bug detection technique. Traditional assertion checking, however, is performed synchronously, imposing its full cost on the runtime of the program. As a result, many useful kinds of checks, such as data structure invariants and heap analyses, are impractical because they lead to extreme slowdowns.

We present a solution that decouples assertion evaluation from program execution: assertions are checked *asynchronously* by separate checking threads while the program continues to execute. Our technique guarantees that asynchronous evaluation always produces the same result as synchronous evaluation, even if the program concurrently modifies the program state. The checking threads evaluate each assertion on a consistent snapshot of the program state as it existed at the moment the assertion started.

We implemented our technique in a system called STROBE, which supports asynchronous assertion checking in both single-and multi-threaded Java applications. STROBE runs inside the Java virtual machine and uses copy-on-write to construct snapshots incrementally, on-the-fly. Our system includes all necessary synchronization to support multiple concurrent checking threads, and to prevent data races with the main program threads. We find that asynchronous checking significantly outperforms synchronous checking, incurring tolerable overheads – in the range of 10% to 50% over no checking at all – even for heavy-weight assertions that would otherwise result in crushing slowdowns.

[*] Deloro fellow

Categories and Subject Descriptors D.2.4 [*Software/Program Verification*]: Assertion checkers; D.1.3 [*Concurrent Programming*]

General Terms Reliability, Performance, Experimentation

Keywords Assertions, concurrent checking, heap snapshot, dynamic analysis, data structure invariants

1. Introduction

Assertions are a widely used technique for program monitoring and bug detection. Assertions are easy to use—programmers add them directly to their code—and they provide a precise and reliable failsafe for critical program properties. The downside is that assertions are checked synchronously, imposing their full cost directly on the runtime of the program. As a result programmers must take care not to add assertions that are too frequent or too costly. This constraint severely limits the kinds of properties that can be checked using assertions, and existing code reflects this limitation: it often contains simple and cheap assertions, such as `assert(p!=null)`, but almost never contains complex and expensive assertions, such as `assert(redblack_invariants(my_tree))`.

This paper presents *Asynchronous Assertions*, a new mechanism that supports efficient checking of complex assertions, including heap properties and data structure invariants. The central idea is to perform expensive checks concurrently in separate threads, allowing the application execution to continue unimpeded. The key problem we solve is how to ensure that asynchronous evaluation produces the same result as synchronous evaluation, even if the application proceeds to modify the program state while the check is running.

Our solution is to run assertion checks on a *snapshot* of the heap as it existed when the assertion was encountered. We introduce techniques to implement the snapshot mechanism efficiently and to ensure the absence of race conditions between the application and the checking threads. No changes are needed to make assertions asynchronous: programmers simply identify the code that implements the assertions, and our system automatically adds instrumentation that reads from the snapshot and handles concurrency.

The programming model for handling asynchronous assertion failures, however, is necessarily different. Although asynchronous assertions produce the same result as synchronous assertions, the result only becomes known at some later point in execution. Therefore we offer two methods for handling failures. The first is completely asynchronous: like a traditional assertion, a failure causes the program to stop, wherever execution happens to be at that point. The second interface is like a future: when an assertion starts it returns a handle that the application code can use to inquire about the result of the assertion, or force synchronous evaluation.

We implemented asynchronous assertions for Java in the Jikes RVM. Our implementation, called STROBE, works with single- and multi-threaded applications and supports multiple concurrent assertions, each with its own snapshot. On suitable hardware, assertions can be evaluated in parallel on separate CPU cores.

Our implementation includes several optimizations that reduce the cost of creating and managing snapshots. First, it uses per-object copy-on-write to snapshot only those objects that the application actually modifies. Second, it never snapshots objects that are new since the start of the most recent check (since they do not exist in any snapshot). Finally, we share copies between snapshots whenever possible.

Since existing programs do not contain suitably complex assertions, we evaluate our system using a combination of micro-benchmarks (adopted from related work), and synthetic assertions running on a more realistic benchmark. The synthetic assertions allow us to thoroughly explore the performance space: we systematically vary the frequency and cost of assertions, and the number of checking threads.

We find that asynchronous evaluation greatly outperform synchronous evaluation, as long as there are enough checking threads to keep up with the rate of new assertions. With sufficient resources our approach supports significantly higher checking workloads for a given runtime overhead budget. Limiting the overhead to 20%, for example, six checker threads can perform checks that would otherwise slow the program by a factor of 2.5 if evaluated synchronously. With a 30% overhead budget, six threads can perform checks that would otherwise slow the program by a factor of 4. With a 40% budget six checker threads can perform checks that would otherwise slow the program by a factor of 6.5. The contributions of this paper are:

- Asynchronous Assertions, a new assertion checking mechanism that evaluates assertions concurrently with the application, while guaranteeing the same results as synchronous evaluation.
- An efficient implementation of our technique in a system called STROBE built on JikesRVM.

```
private boolean isOrdered() {
  Node n = head;
  while (n != null && n.next != null) {
    if (n.data > n.next.data)
      return false;
    n = n.next;
  }
  return true;
}
```

Figure 1. Example assertion: unmodified code to check whether a linked list is ordered.

- An evaluation of STROBE on different usage scenarios: a set of microbenchmarks with data structure invariant assertions, and a well-known benchmark with a synthetic assertion allowing us to systematically explore the performance space. Our results indicate that STROBE works well in practice: it greatly outperforms synchronous checking, and can execute a range of intensive assertion workloads with overheads ranging from 10% to 50%.

2. Overview

In this section, we show how to use STROBE to write and check a data structure invariant assertion asynchronously. We describe the programming model and give code for a small example.

A Use Case Suppose that we are implementing an ordered linked list, and we want to ensure the invariant that for any node in the list, `node.data <= node.next.data`.

To check this invariant synchronously, we would write a checking method that is called before and after every public method invocation on the data structure. This checking method appears in Fig. 1.

Using Asynchronous Assertions With a large linked list, this invariant may take a long time to check, so it is a good candidate for asynchronous checking. Here we show how to convert it to an asynchronous assertion for use in STROBE.

STROBE defines an interface, `StrobeTask`, which the user must implement to issue an asynchronous assertion. `StrobeTask` contains only one method, `compute()`, which the system will call to compute the result of the check. The definition of `StrobeTask` is shown in Fig. 2.

To convert our synchronous ordering check to an asynchronous one, we simply call the synchronous checking method in Fig. 1 from inside the `compute()` method of an inner class that implements `StrobeTask`. We also tag this method with a special annotation, `@ConcurrentCheck`, so the JVM knows it is meant to run concurrently. Code for this inner class is shown in Fig. 2 (we assume that `isOrdered()` is declared inside the inner class). Note that the code inside `compute()` simply calls `isOrdered()`: no changes are needed to execute the check concurrently; the STROBE system automatically handles all synchronization.

```
public interface StrobeTask {
  public boolean compute();
}

private class InvTask implements StrobeTask {
  @ConcurrentCheck
  public boolean compute() {
    return isOrdered();
  }
}
```

Figure 2. Example assertion modified to use the asynchronous assertions `StrobeTask` interface. No changes to the core assertion check `isOrdered()` are necessary.

When using this asynchronous checking mechanism, we must choose what to do if the check fails (i.e, if the list is unordered). STROBE provides two choices for how to handle check failures. One choice is to exit the program immediately and inform the user that the check failed. This is analogous to traditional assertion checking. The semantics are slightly different, however, because the program continues executing asynchronously and progresses beyond the point where the assertion is started. This model is useful in cases where a failed check would not result in catastrophic side effects, but we want the program to halt. In STROBE, this is implemented by issuing the check using the `StrobeAssertion` class (see Fig. 3).

However, if it *is* crucial that the program not progress beyond a certain point if the check fails, STROBE provides a future-like mechanism for the program to wait for the result of the asynchronous check before proceeding. When the check is issued, STROBE returns a handle to the result of the check, and calling the `get()` method of the handle blocks the program until the result becomes available. Thus the user can separate the initiation of the check from the point when the result is needed, allowing the system to exploit as much concurrency as possible. In STROBE, this is implemented by issuing the check using the `StrobeFuture` class (see Fig. 3).

Note that in either case, the program is allowed to modify the data structure as soon as the check is *issued*, even if the check has not *completed* yet. Because of STROBE's snapshotting mechanism, the check will return the same result as if the data structure had not been modified.

3. Snapshot semantics

In this section we describe the semantics of our snapshot mechanism. We present an abstract model of snapshots and use this model to describe our snapshot algorithms for both single and multiple concurrent assertions. We show how these algorithms capture the state of the program at a particular point without copying all of the objects. We also enumerate several optimizations that further reduce the amount of state we need to preserve and cost of copying. We add synchronization where necessary to avoid race conditions between the application and the asynchronous assertions. Using our approach, we can safely construct snapshots on-the-fly: the assertion code computes its result on a snapshot *while* the application mutates the state.

```
StrobeAssertion sassertion =
  new StrobeAssertion(new InvTask());
sassertion.go();
```

(a)

```
StrobeFuture sfuture =
  new StrobeFuture(new InvTask());
sfuture.go();
...
assert(sfuture.get());
```

(b)

Figure 3. Using (a) `StrobeAssertion` and (b) `StrobeFuture` to issue the check. With `StrobeAssertion`, the program will exit immediately if the check returns false. With `StrobeFuture`, the system returns a handle to the future result, and at a crucial point, the program can block and wait for the result by calling `get()`.

3.1 Snapshot model

A snapshot captures the state of the program at a moment in time. Since our goal is to support complex checks, such as data structure invariants, our focus is on snapshotting the state of the heap. The heap consists of a set of objects containing both primitive values as well as references to other objects. We assume that objects are mutable, so it is necessary to preserve object states in order to construct a snapshot.

We model heap mutability as a series of object versions: conceptually, each time an object is modified a new version is created. Given an object o, o_i represents the state of the object after i mutations. The most recent version is represented by the mapping $current(o)$. In the application's code, a field access always retrieves its value from the current (most recent) version: `o.f` is interpreted as $current(o).f$.

A snapshot, then, is the set of object versions that were current at the moment the snapshot is requested (in our case, the moment an asynchronous assertion is started). Our algorithm is based on the observation that many of these versions will *continue* to be current, as long as the application does not modify them. The only object versions we need to preserve are those that the application mutates after the start of the assertion check. In the discussion below we refer explicitly to versions of objects, but in our implementation we preserve an object's state by copying it right before the write occurs (copy-on-write). By making this operation atomic, an assertion can safely access its snapshot, even if the application changes the heap concurrently.

3.2 Single snapshot

We start with an algorithm that computes a single heap snapshot for a single asynchronous assertion. Initially, there are no preserved versions of objects. Evaluating the assertion in this state is equivalent to traditional synchronous evaluation. When an asynchronous assertion is encountered the system enables the following read and write behavior by atomically setting a global flag $active$ to true.

The version of an object we want to preserve is the version that was current when the snapshot started. This version will still be the current version at the *first* write to the object (if any write occurs). So, at an application write we first check to see if the given object already has a preserved version in the snapshot. If so, there is no extra work to do. If not, we keep a reference to that old version in a separate mapping called $preserved$ before allowing the write to proceed and produce a new version. A Boolean flag called $modified$ records which objects have been preserved.

At application write to object o:

```
if $active$ and $!modified(o)$
  $preserved(o) := current(o) = o_i$
  $modified(o) :=$ true
  write to $o$ , $current(o) := o_{i+1}$
```

The assertion checking code accesses the snapshot by first determining whether to use the original object or the preserved version. A read of the form `o.f` is interpreted $snapshot(o).f$ where:

$$snapshot(o) = \begin{cases} preserved(o), & modified(o) \\ current(o), & \text{otherwise.} \end{cases}$$

When an assertion completes, the assertion code sets the $active$ flag to false, and clears the $preserved$ and $modified$ data structures.

3.3 Guarantees

We guarantee that when the assertion code reads an object it sees either (a) the original object, if it has not been modified since the time the check started, or (b) a copy of the object that reflects its state at that time, if it has been modified.

In our implementation we ensure the absence of race conditions by updating $modified$ and $preserved$ atomically. No interleaving allows the assertion checker to see modifications to the object that occur after it starts. Our synchronization strategy is described in more detail in Section 4 (Implementation). Note, however, that there *can* be races when multiple application threads start assertions without synchronizing with each other.

Notice in the algorithm above that assertion checking code applies the $snapshot$ mapping at every field load. This algorithm avoids exposing direct references to preserved versions of objects, which is critical for both correctness and performance:

- *Snapshots change dynamically.* An object write in the application can occur in the middle of an assertion check, triggering a snapshot operation. Earlier assertion code might have read from the original object, but all subsequent accesses must be redirected to the snapshot.
- *Object identity is preserved.* Hiding snapshot references is crucial for the correctness of code that relies on object identity (such as reference comparison and `IdentityHashMap`, which could be destroyed if we allow a mix of references to the current and old versions of an object.
- *No reference "fixup".* We never need to fix object references in the running assertion code, since snapshot references are never held directly in local variables or stored in any data structures.

3.4 Multiple snapshots

Multiple concurrent asynchronous assertions can be triggered either by a single thread issuing one assertion before another is completed, or by multiple application threads issuing assertions concurrently. In either case, each concurrent assertion needs its own snapshot, corresponding to the heap state at the time it started.

The main problem we solve is determining, at a given application write, which snapshots need to preserve the current object version and which do not. Again, the key observation is that we only need to preserve an object the first time it is modified with respect to any given snapshot.

We start by introducing an *epoch* counter E: E is initially 0; the system increments E each time it starts executing a new asynchronous assertion. The epoch number serves as a unique identifier for each dynamic check instance. The flag $active$ becomes a set of flags, one for each epoch, that indicates which checks are still running.

Instead of simply recording whether an object is modified or not, we record *in which epoch* it was last modified. We store this information in a mapping $modifiedAt(o)$. An object needs to be preserved for an assertion started at time E_t if it was last modified in an epoch *before* E_t. Finally, we augment the $preserved$ mapping to hold multiple object versions, one for each snapshot.

At an application write to object o:

```
for each assertion $E_t$
  if $active(E_t)$ and $modifiedAt(o) < E_t$
     then $preserved(o, E_t) := current(o) = o_i$
  $modifiedAt(o) := E$
  write to $o$ , $current(o) := o_{i+1}$
```

Each assertion reads from its snapshot, as necessary. A read of the form `o.f` in an active assertion instance E_t is interpreted $snapshot(o, E_t).f$ where:

$$snapshot(o, E_t) = \begin{cases} preserved(o, E_t), & modifiedAt(o) \geq E_t \\ current(o), & \text{otherwise.} \end{cases}$$

3.5 Optimizations

The algorithms above construct relatively small snapshots, consisting of only those objects that the application modifies. We further reduce the cost of constructing snapshots using two optimizations:

- **Skip new objects:** We never need to preserve newly created objects. Objects created since the start of the most recent asynchronous assertion do not exist in any snapshot. We can easily implement this optimization in our framework by initializing $modifiedAt(o)$ to the current epoch value E for any newly allocated objects. This optimization proves extremely valuable because it prevents unnecessary copying that would otherwise be triggered by frequent mutations in object constructors.
- **Share object copies:** At any given write, the snapshots that need to preserve the mutated object all need the same version of that object (i.e., the current version). Since our implementation uses copying to preserve state, we can share one copy across all the snapshots that need it. This optimization also allows us to simplify the test for copying: at a write if $modifiedAt(o) < E$ then at least one snapshot needs a copy of object o.

4. Implementation

In this section, we describe the implementation of the algorithms presented in Section 3. The two main challenges are (a) finding an efficient way to store and access snapshots, and (b) ensuring that snapshot management is free of data races. The system is implemented in Jikes RVM 3.1.1, the most recent stable release, and consists of three major components:

- Copying write barrier: When an assertion starts, STROBE activates a special write barrier in the application code that constructs a snapshot of the program state. It copies objects as necessary to preserve the state and synchronizes snapshot access with the checker threads.
- Checker thread pool: Checker threads pick up assertions as they are issued and execute them. If all checker threads are busy, assertions block the application thread until one is free.
- Snapshot read barrier: Assertion checking code is written in regular Java, tagged with an annotation that the compiler recognizes (see Fig. 2). The code is compiled with a read barrier that returns values from object snapshots whenever they are present.

4.1 Snapshot storage and management

The primary goal of our design is to minimize the impact of snapshot management on the performance of the application threads. The last modification time of an object (the $modifiedAt$ value, expressed in terms of epoch number)

```
void writeBarrier(Object src, Object target,
                  Offset offset)
{
  int epoch = Snapshot.epoch;
  if (Header.isCopyNeeded(src, epoch)) {
    // -- Needs to be copied, we are the copier
    //    timestamp(src) == BEING_COPIED
    snapshotObject(src);
    // -- Done; update timestamp to current epoch
    Header.setTimestamp(src, epoch);
  }
  // -- Do the write (omitted: GC write barrier)
  src.toAddress().plus(offset).store(target);
}
```

Figure 4. Copy-on-write write barrier

```
boolean isCopyNeeded(Object obj, int epoch)
{
  int timestamp;
  do {
    // -- Atomic read of current timestamp
    timestamp = obj.prepareWord(EPOCH_POS);
    // -- If in current epoch, nothing to do
    if (timestamp == epoch)
      return false;
    // -- If someone else is copying, wait
    if (timestamp == BEING_COPIED)
      continue;
    // -- ...until CAS BEING_COPIED succeeds
  } while (!obj.attempt(timestamp, BEING_COPIED,
                        EPOCH_POS));
  return true;
}
```

Figure 5. Snapshot synchronization

must be checked at every single write. To make this operation fast we store the timestamp information in an extra word added to the header of each object.

Preserving the state of an object is accomplished by taking a snapshot of it right before a write. Because this copy operation occurs in the application thread, we put significant effort in reducing the cost of copying and eliminating unnecessary copies. Since the original object is mutated, all existing references to it continue to point to the most current version. No extra work is needed to maintain this information (i.e., the $current(o)$ mapping is a no-op.)

Since we have a fixed pool of checker threads, much of the information about active assertions is identified by the ID of the checker thread, not by the epoch number. With T checker threads we will have at most T simultaneous snapshots, and therefore at most T copies of any given object. We store this information (the $preserved$ mapping) in a per-object array of T references (one for each potential snapshot), indexed by thread ID. We refer to it as the *forwarding array*.

4.2 Synchronization

All snapshot operations, both reads and writes, work on a single object at a time. Therefore we can synchronize these operations on the $modifiedAt$ word in the object header. We check and update this value using only atomic operations, and use two techniques to avoid race conditions.

First, the write barrier stores a “being copied” sentinel value in the $modifiedAt$ word during copying. The sentinel value ensures that three operations—copying the object, updating the epoch timestamp, and performing the write—occur as an atomic unit. This avoids a potential race condition in which two application threads modify the same object, resulting in two snapshots, one of which is incorrect. Note that this situation does not necessarily represent a race in the application if the two threads are updating different fields. Without synchronization it is possible for both threads to determine that the object must be copied (both see $modifiedAt(o) < E$). One thread copies the object, installs the copy in the forwarding array, and performs its write. The second thread then copies the object again—but this copy includes the write from the first thread, which should not be visible to the checker.

Second, we order these operations in such a way that the checker thread cannot see intermediate results: the write barrier only updates the timestamp after it has made the copy, but before it applies the write. The read barrier might access the copy before it needs to, but it will never see new values written to the object. To be safe, we also fully copy the object before installing its reference in the forwarding array.

4.3 Write barrier

The write barrier is shown in Fig. 4 (slightly simplified from the actual code). All operations on the forwarding array (making or accessing copies) are synchronized using atomic operations on the object’s timestamp. The write barrier first calls a method to determine if a copy is needed. The method `isCopyNeeded()` is shown in Fig. 5. It consists of a loop that exits when either (a) the timestamp is current, so no copy is needed, or (b) the timestamp is older than the current epoch, so a copy is needed. In case (b) the code writes a special sentinel value `BEING_COPIED` into the timestamp, which effectively locks the object. All other reads and writes are blocked until the sentinel is cleared. This code is compiled inline in the application.

4.4 Snapshot creation

The snapshot code, shown in Fig. 6 (slightly simplified from the actual code), is compiled out-of-line, since it is infrequent and relatively expensive. It first loads the forwarding array, creating one if necessary. It then makes a copy of the object using an internal fast copy (the same mechanism used in the copying garbage collectors). It installs a pointer to the copy in each slot of the forwarding array for which the corresponding checker thread is (a) active and (b) has not already copied the object (slot is null).

Our system allocates all object copies and forwarding arrays in the Java heap, which allows them to be automatically reclaimed by the garbage collector. Our system is completely independent of any specific garbage collection algorithm, we only require the garbage collector to consider forwarding arrays in its processing of the heap (this is the only change we made to the collector). When an assertion completes, the checker nulls out all of the slots in the forwarding arrays corresponding to its thread ID, eliminating the only references to the snapshot objects. Forwarding arrays are reclaimed when the original objects become garbage.

```
void snapshotObject(Object obj)
{
  // -- Get forwarding array; create if needed
  Object[] forwardArr =
               Header.getForwardingArray(obj);
  if (forwardArr == null) {
    forwardArr = new Object[NUM_CHECK_THREADS];
    Header.setForwardingArray(obj, forwardArr);
  }
  // -- Copy object
  Object copy = MemoryManager.copyObject(obj);
  // -- Provide copy to each active checker
  //    that has not already copied it
  for (int t=0; t < NUM_CHECK_THREADS; t++) {
    if (isActiveCheck(t) &&
        forwardArr[t] == null)
      forwardArr[t] = copy;
  }
}
```

Figure 6. Copying code

```
// -- Returns the object to read from,
//    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;
    else {
      // -- ...otherwise return copy (snapshot)
      return copy;
    }
  }
}
```

Figure 7. Read barrier

4.5 Read barrier

Assertion checking code is compiled with a read barrier, shown in Fig. 7 (also slightly simplified), that accesses the forwarding array as necessary to read data from object snapshots. The read barrier first loads the forwarding array, and if the array is non-null, it uses the ID of the checker thread to index into the forwarding array and retrieve the copy belonging to that snapshot. If either the forwarding array or the copy is null, it indicates that the object has not been copied, and the read barrier returns the current reference to the object, which may then be read.

5. Experimental Evaluation

In this section we present an experimental evaluation of STROBE. The focus of this evaluation is not on bug detection: our technique detects all the same errors that traditional synchronous assertions would catch. Our goal is to explore the performance space of this technique in order to provide a sense of how well it works under a range of conditions. Our main findings are:

- Asynchronous checking performs significantly better than synchronous checking in almost all circumstances. Only when individual checks are extremely brief does the overhead of concurrency overwhelm the benefit.
- Since assertions are independent, they parallelize perfectly. As long as there are enough checker threads to keep up with the demand, our technique slows overall runtime by 10% to 60%, depending on the assertion workload. By comparison, evaluating the same assertions synchronously slows runtime by 1.25X to 8X.
- When there are not enough checker threads for the assertion workload, the application must often wait for an available checker. At this point the slowdown begins to grow quicky, at a rate similar to synchronous evaluation.
- The main source of overhead is creating and maintaining snapshots. We found, however, that up to 1/3 of the overhead is due to other factors – a combination of the cost of the extra words in the object header (one for the epoch, and one for the forwarding address) and possibly additional pressure on the memory system.
- Sharing object copies between snapshots significantly reduces overhead. Without this optimization, snapshot overheads would be approximately 50% higher.

5.1 Experimental set up

We evaluate STROBE using two kinds of experiments: microbenchmarks instrumented with data structure invariant checks, and a real benchmark program (SPEC JBB2000), instrumented with synthetic assertions that allow us to systematically vary the frequency and cost of the checks. Both sets of benchmarks are assertion-intensive, making them a challenge to execute efficiently.

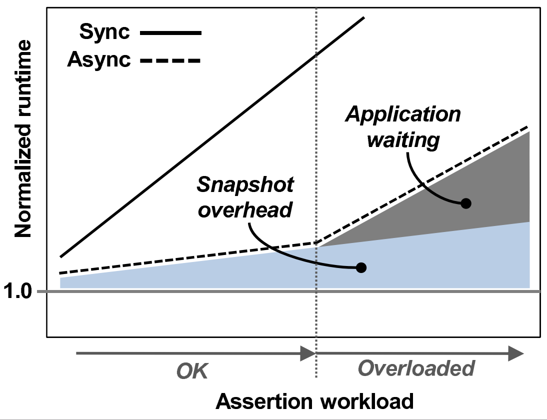


Figure 8. Typical results: Asynchronous evaluation has a low overhead as long as the assertion workload does not overwhelm the checker threads.

```
int traverseAndCheck(Node n, Node p)
{
    if (n == null)  return 1;
    Node l = n.left;
    Node r = n.right;
    // -- Recursive traversal:
    //     get num of black nodes below us
    int lb = traverseAndCheck(l, n);
    int rb = traverseAndCheck(r, n);
    // -- Check that the tree is balanced
    if (lb != rb || lb == -1)  return -1;
    // -- Check that the tree is ordered
    Integer val = (Integer) n.key;
    if (l != nil && val <= (Integer) l.key)
      return -1;
    if (r != nil && val >= (Integer) r.key)
     return -1;
    // -- Check colors:
    int c = n.color;
    if (c == RED &&
        (l.color != BLACK || r.color != BLACK))
      return -1;
    // -- Return total num black nodes
    return lb + (n.color == BLACK ? 1 : 0);
}
```

Figure 9. An assertion procedure that performs recursive checking of various safety properties on a red-black tree.

Microbenchmarks. Our microbenchmarks experiments replicate the evaluation presented in the Ditto work [13]. We implemented two data structures: an ordered linked list and a tree-map using a red-black tree. For each one, we added a method that checks the data structure invariants:

- Ordered linked list: check that `link.next.prev == link` and that elements are ordered.
- TreeMap red-black tree: (a) make sure values are in order, (b) check that all children of red nodes are black, and (c) make sure that all paths from the root to a leaf have the same number of black nodes. The code is shown in Fig. 9.

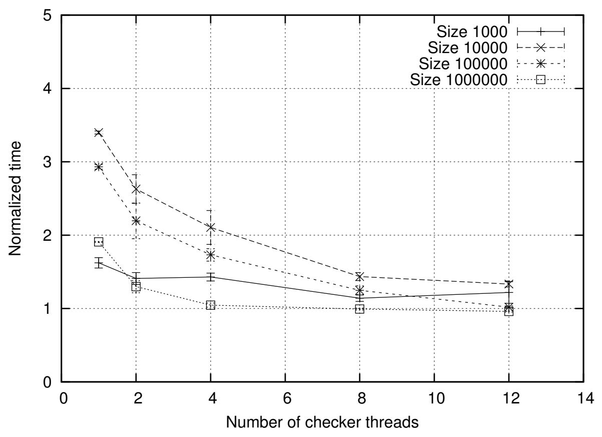


Figure 10. Ordered linked list: overhead versus number of checker threads for various list sizes. The checks are too brief to recoup the cost of synchronization and snapshotting.

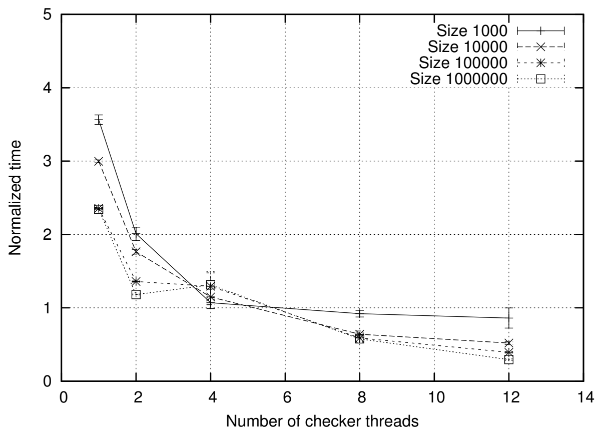


Figure 11. TreeMap (red-black tree): overhead versus number of checker threads for various tree sizes. With more than five checker threads, asynchronous checks beat synchronous checks.

SPEC JBB2000 SPEC JBB2000 [14] is a multithreaded benchmark that emulates a three-tier client-server system, with the database replaced by an in-memory tree and clients replaced by driver threads. The system models a company, with warehouses serving different districts and processing customer orders. In a single run, the benchmark executes 70,000 transactions against its database. The runtime is 1.99 seconds in a completely unmodified JikesRVM (no assertions, no extra header words) running on our hardware.

We added a synthetic assertion check on the main Company data structure right before each transaction is processed (in `TransactionManager.go()`). This synthetic assertion allows us to systematically vary the frequency of assertions and the amount of work performed by each check. This assertion performs a bounded transitive closure on the object it is given, emulating a fixed amount of data structure traversal and access. We control the frequency of these assertions by selecting some fraction of them to perform.

Methodology We compare three kinds of runs: *asynchronous assertions* (with varying numbers of checking threads), *synchronous assertions* (stop and evaluate each assertion), and *baseline* (no assertions, no RVM modifications). Our primary measurement is overall application runtime. For asynchronous checks we also measure the number of objects copied during the run and the total number of bytes copied. Our system is GC-algorithm independent, and all experiments use the generational mark/sweep collector.

For the microbenchmarks we run each test 20 times and compute the average and confidence interval. For the SPEC JBB experiments we run four iterations of the benchmark and time the last iteration. The differences we observe in runtimes between synchronous and asynchronous checking are so great that small perturbations in the execution do not affect our overall results. We fix the Java heap size to 120MB, which is approximately 2X the minimum.

All experiments were run on a 12-core machine (dual six-core Xeon X5660 running at 2.8GHz) with 12GB of main memory running Ubuntu Linux kernel 2.6.35. We limit the number of checker threads to 10, since we need 1 or 2 cores to run the application threads.

5.2 Results: Microbenchmarks

Each run of the microbenchmark program builds one of the two data structures of a specific size and performs 1000 operations on it. Each operation is either an *add*, a *remove*, or an *access*. We perform the invariant checks before and after each add or remove. Operations are chosen at random so that 90% are accesses, 9% are adds, and 1% are removes.

For each data structure, we ran experiments with size 1,000, 10,000, 100,000 and 1,000,000 elements. We varied the number of checker threads from 1 to 12. The results are presented in Figures 10 and 11. Time is normalized to the cost of the synchronous checks.

We find that for the list, the cost of the asynchronous mechanism (coordinating with the checker threads and building the snapshots) overwhelms the relative simplicity of the computation. While performance improves with more threads, we never significantly improve upon synchronous checking. For the TreeMap, however, increasing the number of threads improves performance substantially, particularly as the size of the data structure increases. For all input sizes, we hit the break-even point at around five threads; using 12 threads reduces the overhead of checking to within a small fraction of the baseline time for all but the smallest input.

5.3 Results: SPEC JBB

Our synthetic transitive closure assertion allows us to explore the performance space of our technique by systematically varying the assertion workload along two axes: frequency of checks and cost of each check:

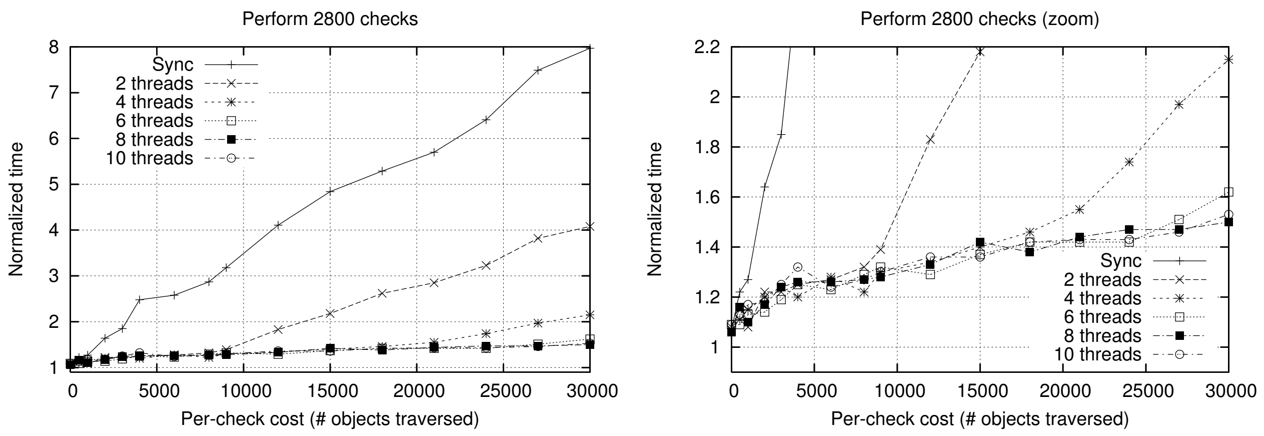


Figure 12. SPECJBB: Overhead of a fixed number (2800) of assertions for a range of costs, normalized to baseline (no assertions, unmodified RVM). The overhead of asynchronous evaluation grows slowly as long as the checker threads is adequate for the assertion workload. Both graphs show the same data; right graph zooms on the region where the threads are not saturated.

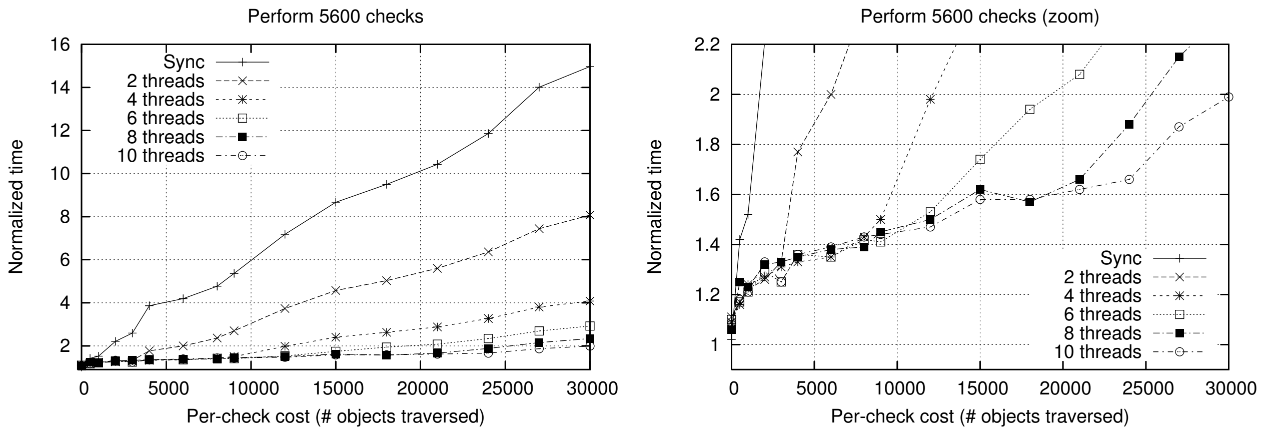


Figure 13. SPECJBB: Overhead of a fixed number (5600) of assertions for a range of costs, normalized to baseline (no assertions, unmodified RVM). Left graph shows all data; right graph highlights the unsaturated region (1X to 2.2X).

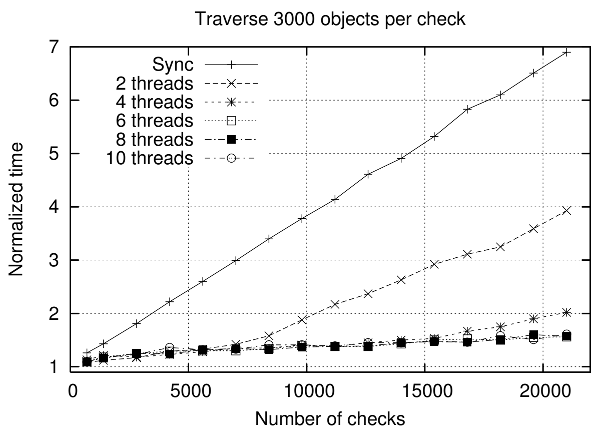
- **Frequency:** We vary the number of checks from 700 per run (1 every 100 transactions) to 21,000 per run (1 every 3 transactions).
- **Cost:** We vary the cost by varying the fraction of the database that each assertion traverses, measured in number of objects visited – from 1 object to 30,000 objects.

We run the program with synchronous checks and with asynchronous checks using 2, 4, 6, 8, and 10 checker threads. All times are normalized to the runtime of SPEC JBB with no assertions running on a completely unmodified JikesRVM. This baseline run takes about about 1.99 seconds on our hardware configuration. Because these experiments generate so much data we present *slices* in which we fix one axis of the workload (frequency or cost) and vary the other.

Fix frequency, vary cost. Fig. 12 and Fig. 13 show the relative slowdown for a fixed number of assertions (fixed frequency) over a range of assertion costs, comparing synchronous checking against asynchronous checking with 2 to 10 threads. Each figure consists of a pair of graphs of the same data: the graph on the left shows all the data, the graph on the right zooms in on a range of the Y axis from 1 to 2.2. Both figures show the same trends:

- The overhead of synchronous checking grows very rapidly as the cost of the assertions (size of the traversal) increases from 1 object to 30,000 objects.
- The overhead of asynchronous checking grows relatively slowly until the checker threads become saturated, at which point the overhead begins growing rapidly because the application must wait for an available checker – the performance curve turns sharply up, with a slope similar to the synchronous configuration.

With a workload of 2800 assertions (Fig. 12), 2 checker threads become saturated at an assertion cost of 8000, while 4 threads become saturated around 22,000. With 5600 asser-



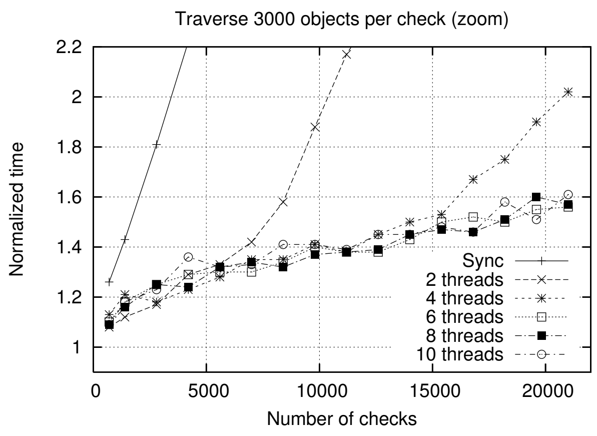
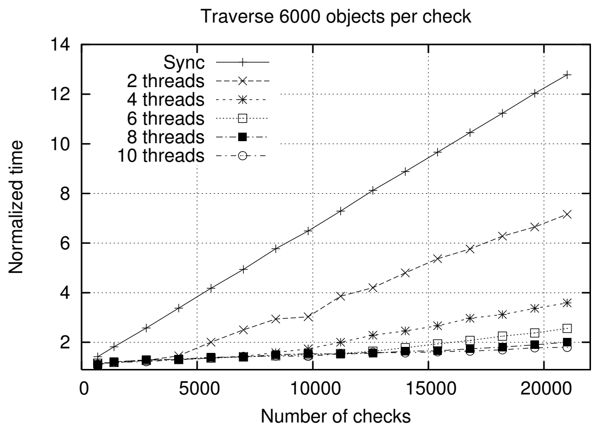


Figure 14. SPECJBB: Overhead of fixed-cost (traverse 3000 objects) assertions, over a range of assertion frequencies, normalized to baseline (no assertions, unmodified RVM). Right graph zooms in on the unsaturated region (1X to 2.2X)



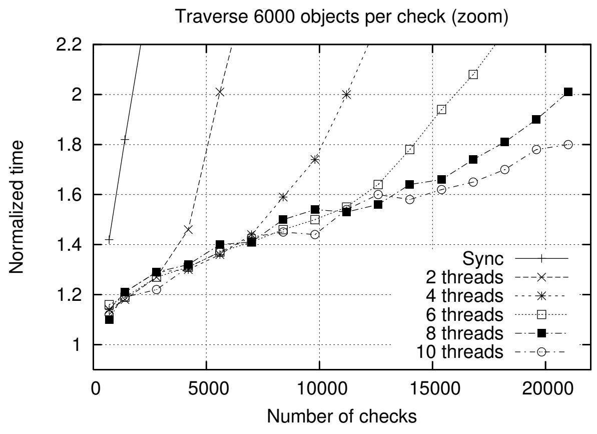


Figure 15. SPECJBB: Overhead of fixed-cost (traverse 6000 objects) assertions, over a range of assertion frequencies, normalized to baseline (no assertions, unmodified RVM). Right graph zooms in on the unsaturated region (1X to 2.2X)

tions (Fig. 13), all configurations eventually become saturated when the cost of each assertion exceeds 25,000 objects.

The zoomed versions of these graphs (on the right) highlight the configurations where the threads are not overloaded, and the main source of overhead is the snapshot mechanism. With 2800 assertions the overhead grows from less than 10% to around 45%. With 5600 assertions the overhead grows from 10% to 60% at the saturation point.

Fix cost, vary frequency. Fig. 14 and Fig. 15 show the relative slowdown for a fixed cost assertion (size of heap traversal) over a range of frequencies, comparing synchronous checking against asynchronous checking with 2 to 10 threads. As above, each figure consists of a pair of graphs of the same data: the graph on the left shows all the data, the graph on the right zooms in on a range of the Y axis from 1 to 2.2. These graphs show the same trends as the fixed-frequency graphs:

- 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.

With a 3000-object traversal (Fig. 14), 2 checker threads become saturated at a frequency of 7000 assertions, while 4 threads become saturated around 15,000 assertions. With a 6000-object traversal (Fig. 15), all configurations become saturated by the time we have 15,000 assertions, although 10 threads manage to keep the overhead under 80%.

In the zoomed versions of these graphs (on the right), notice that increasing the frequency of assertions causes overhead to grow more quickly than increasing the cost of each assertion (as shown in Fig. 12 and Fig. 13). The reason for this difference is that there is a per-assertion cost associated with initiating and cleaning up a heap snapshot.

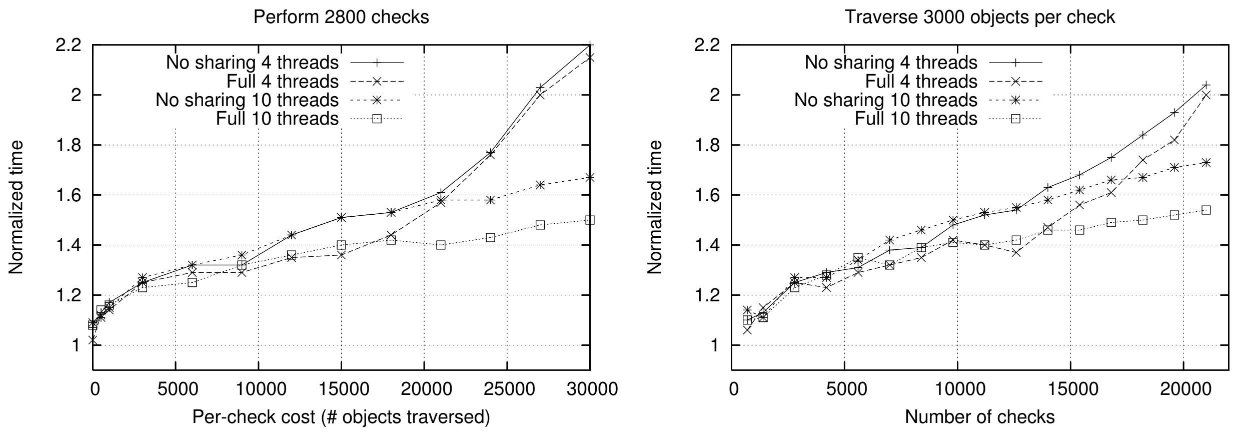


Figure 16. SPECJBB Snapshot sharing: When snapshots do not share object copies, overhead increases by 25% to 30% over the unmodified (Full) STROBE implementation (with the sharing optimization).

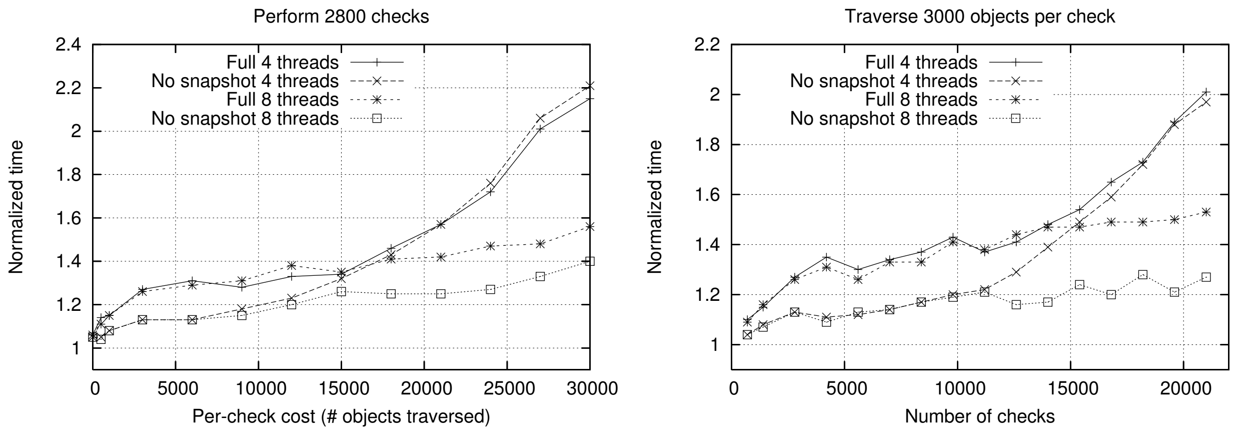


Figure 17. SPECJBB Lower bound: Evaluate asynchronous assertions without creating snapshots. Creating and maintaining snapshots accounts for approximately 2/3 of the overhead.

Effect of not sharing copies. One of the consequences of our snapshot algorithm is that in many cases multiple snaphots can share a single copy of an object. When an object is modified, any assertion that started since the previous modification needs the same image of that object. To measure the benefit of this optimization we turned it off and compared this no-sharing version to the full version of STROBE. Fig. 16 shows these results for a fixed frequency (2800 checks, on the left) and a fixed cost (3000 object, on the right). As the assertion workload increases, the benefit of sharing copies becomes significant: sharing copies reduces overhead by 25% to 30% – e.g., from an overhead of 70% (no sharing) to an overhead of 50% (with sharing).

Lower bound: no snapshots. In order to explore the lower bound of this technique we use a version of STROBEthat performs asynchronous checking *without* heap snapshots. This version does not produce correct results (since assertions see the changing state of the heap), but allows us to eliminate the overhead of creating and maintaining snapshots. Fig. 17 shows the results for a fixed frequency (2800 checks, on the left) and a fixed cost (3000 object, on the right). Eliminating the snapshot reduces the overhead significantly, but not completely. We believe that two other factors contribute to overhead. First, we compare our system against a baseline that does not include extra header words – eliminating these words reduces GC load. Second, asynchronous assertions place additional demands on the machine's memory system, including the caches and memory bandwidth.

Copying costs and GC time. During each asynchronous run we count the total number of objects copied and the volume in bytes for all snapshots. Fig. 18 shows the volume of bytes copied as a function of assertion frequency. This cost ranges from 25,000 objects (2MB) for fast, infrequent checks up to 800,000 objects (70MB) for long-running and frequent checks. Note that the graph flattens out as the number of checks increases. The reason is that under heavier assertion workloads there are evaluations in-flight at all times, so all writes to the heap are generating object copies.

Since object copying occurs in the write barrier in the application thread, it directly impacts overall runtime. In addition, the extra objects reside in the Java heap, increasing garbage collection time proportionally. On average, GC time increases by 10% to 50% over synchronous evaluation. In our configuration, however, GC time only accounts for 5% to 10% of total runtime, so the impact is low. Under tighter memory constraints the extra memory used by snapshots would likely have a bigger impact on runtime. In our lower bound experiment (described below) we turned off snapshots to measure this cost.

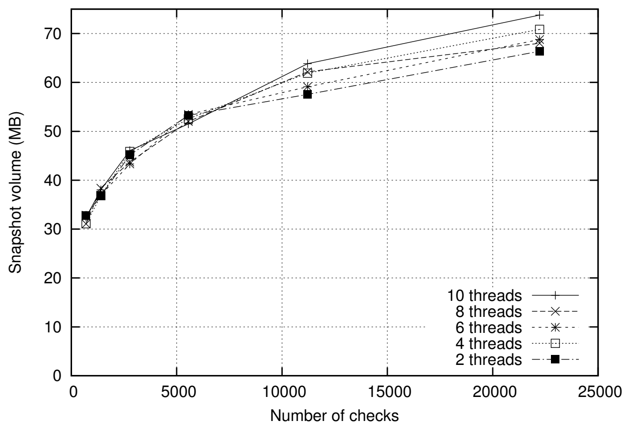


Figure 18. SPECJBB copying costs: more checks require more snapshots, increasing the volume of bytes copied, but tops out as all heap writes generate copies.

6. Related Work

Our work is related to a large body of previous work on program checking. In this section, we briefly discuss some of the related work on dynamic checking, usage of snapshots in concurrency, and other related runtime techniques such as futures, concurrent garbage collection and software transactions. No existing technique, however, supports complex dynamic checks evaluated at a specific point in the program with low overhead and high accuracy.

Future contracts Our work is closely related to recent work on *future contracts* [7], a technique for concurrently checking behavioral software contracts in Scheme programs. The authors propose (but only partially implement) a very different approach for handling mutations in the store. They view a write by the main thread as a kind of synchronization, which blocks until the checking thread has had a chance to read the old value (in effect, blocking the main thread until the check completes). This approach is unlikely to perform well for imperative languages, where mutations are frequent. Our approach is to copy the old state, allowing the main thread and the checker thread to proceed without blocking. Our infrastructure could be used for a more complete, high-performance implementation of future contracts that supports imperative languages, avoids unnecessary synchronization, and supports multiple concurrent assertions.

Dynamic program checking Dynamic analysis avoids the main problem with static analysis: by performing error checks on the actual concrete heap of a running program, it does not have to make conservative assumptions about potential program state. Our system addresses the primary challenge of dynamic checking: runtime overhead. There are a number of existing approaches for reducing the cost of dynamic checks. These techniques are orthogonal to ours, however: any of them could be combined with our system to obtain the benefits of both.

The Ditto system speeds up invariant checking by automatically incrementalizing the checking code [13]. The incremental checks are still quite expensive, however, and since they are run synchronously, significantly slow program execution. The QVM system provides a number of *heap probes* to check data structures, and manages the cost by reducing the frequency of checking [2]. The Phalanx system attempts to reduce the cost of assertion checking by parallelizing the queries. In both QVM and Phalanx, the queries are evaluated synchronously with respect to the application.

Recent work has explored the idea of piggybacking heap checks on the garbage collector. In the work by Aftandilian et al, [1], assertions are evaluated synchronously at a regularly scheduled garbage collection. Subsequent work by Reichenbach et. al [12] investigates the class of assertions that can be computed with the same complexity as garbage collection. The two main limitations of this technique are (a) the kinds of checks it supports are limited by the GC algorithm (in particular, the single-touch property), and (b) the frequency of checks is limited because they are evaluated only when a collection occurs.

Concurrent program checking The SuperPin system [17] provides some of the same capabilities as ours but through a completely different mechanism. SuperPin uses the fork system call to create a consistent snapshot, which it uses to run an instrumented copy of the main program. As with other previous work, however, this system is focused on relatively low-level program analysis added as binary instrumentation, and performance is still too slow for production use (100-500% slowdown).

Speck [11], Fast Track [10], and ParExC [15] are all speculative execution systems. Speck and ParExC focus on low-level runtime checks such as array out-of-bounds and pointer deference checks. Fast Track enables speculative unsafe optimizations, whose correctness it checks by the unoptimized program on multiple processors. None of these systems are designed to check the high-level program properties we are interested in.

Safe Futures for Java [18] provides a safe implementation of futures in Java. They handle concurrency issues between the main program and the futures via many of the same mechanisms as our work: read and write barriers and object versioning. However, their system does not work with multithreaded programs, and because their futures can write to the heap, they must use a read barrier for all memory accesses. Our assertions cannot write to objects in the snapshot, so we can avoid a read barrier in the program code and achieve better performance.

Transactional memory Recently, there has been significant amount of work on transactional memory [8]. In principle, it is possible to treat each assertion evaluation as a separate transaction, and let the STM detect conflict. However, the problem with this approach would be that either the assertion or the program would be rolled back every time a conflict occurs, which is highly likely for long-running assertions that may touch large portions of the heap. Frequent rollbacks would render the system unusable (and restarting the application has its own issues when it comes to side effects such as exceptions and I/O). Further, with our semantics, rollbacks are unnecessary, as the assertion only needs to compute a result with respect to the state in which it was triggered.

Concurrent Programming Models Recently, there has been an increased interest in new concurrent programming models [3, 6]. Essentially, in these models, revisions are used to ensure that each process operates on its own local snapshot. When processes complete, their local snapshot changes are merged into the global state, or if their changes conflict (i.e., if they modify the same memory location), either a conflict is declared and the program aborts [3] or in the case of more advanced conflict resolution strategies, the programmer can provide a merge function which is used to resolve these conflicts [6]. We can think of our asynchronous assertion model as a restricted form of concurrent revisions where the asynchronous processes (the assertions) do not modify the heap and hence there is no need to perform conflict checking.

Snapshot-based concurrent garbage collectors Copy-on-write techniques for obtaining snapshots have seen extensive use in mark-and-sweep, snapshot-at-the-beginning concurrent garbage collectors, i.e., [4, 5, 9, 16, 19]. Once the garbage collector starts working, it computes all reachable nodes in the heap *at the time the collector started* by operating on a snapshot maintained by a write barrier, as in our system. There are a few basic technical differences between snapshot collectors and this work. First, there is no need for multiple garbage collectors to be running at the same time computing the same information. Usually, a new collector cycle is started after the previous one has ended. In contrast, it is sensible to have multiple assertions at the same time, and hence our system must support this scenario. Second, collectors are computing a specific property: transitive closure from a set of roots. In our case, each snapshot can be used to compute completely different assertions. Third, with concurrent collectors, the program needs to intercept only heap reference modifications, while in our case, depending on whether the assertion accesses primitive values in the heap, it may be desirable to snapshot their modification as well.

7. Conclusions

Assertions are a powerful and convenient tool for detecting bugs, but have always been limited by the fact that the checking cost is paid for in application runtime. In this paper we show how assertions can be checked asynchronously, greatly reducing checking overhead. Our technique enables a much wider range of assertions, including complex heap checks and data structure invariants. We believe that this approach will become even more appealing as modern processors continue to add CPU cores, providing ample additional computing power that can be devoted to making software run more reliably.

Acknowledgments

This work is supported by the National Science Foundation under grant CCF-1018038 and by IBM. Opinions, interpretations, conclusions, and recommendations expressed in this material are those of the authors and do not necessarily reflect those of the sponsors.

References

[1] E. Aftandilian and S. Z. Guyer. GC Assertions: Using the Garbage Collector to Check Heap Properties. In *ACM Conference on Programming Languages Design and Implementation*, pages 235–244, 2009.

[2] M. Arnold, M. Vechev, and E. Yahav. QVM: An Efficient Runtime for Detecting Defects in Deployed Systems. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 143–162, 2008.

[3] A. Aviram, S.-C. Weng, S. Hu, and B. Ford. Efficient System-Enforced Deterministic Parallelism. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, pages 1–16, 2010.

[4] H. Azatchi, Y. Levanoni, H. Paz, and E. Petrank. An On-the-fly Mark and Sweep Garbage Collector Based on Sliding Views. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 269–281, 2003.

[5] D. F. Bacon, P. Cheng, and V. T. Rajan. A Real-time Garbage Collector with Low Overhead and Consistent Utilization. In *ACM Symposium on the Principles of Programming Languages*, pages 285–298, 2003.

[6] S. Burckhardt, A. Baldassin, and D. Leijen. Concurrent Programming with Revisions and Isolation Types. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 691–707, 2010.

[7] C. Dimoulas, R. Pucella, and M. Felleisen. Future Contracts. In *Proceedings of the 11th ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, pages 195–206, 2009.

[8] T. Harris, J. Larus, and R. Rajwar. *Transactional Memory, 2nd edition*. Morgan and Claypool Publishers, 2010.

[9] R. Jones and R. Lins. *Garbage Collection*. John Wiley and Sons, 1996.

[10] K. Kelsey, T. Bai, C. Ding, and C. Zhang. Fast Track: A Software System for Speculative Program Optimization. In *International Symposium on Code Generation and Optimization*, pages 157–168, 2009.

[11] E. B. Nightingale, D. Peek, P. M. Chen, and J. Flinn. Parallelizing security checks on commodity hardware. In *ACM Conference on Architectural Support for Programming Languages and Operating Systems*, pages 308–318, 2008.

[12] C. Reichenbach, N. Immerman, Y. Smaragdakis, E. E. Aftandilian, and S. Z. Guyer. What Can the GC Compute Efficiently?: A Language for Heap Assertions at GC Time. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 256–269, 2010.

[13] A. Shankar and R. Bodík. DITTO: Automatic Incrementalization of Data Structure Invariant Checks (in Java). In *ACM Conference on Programming Languages Design and Implementation*, pages 310–319, 2007.

[14] *SPECjbb2000 Documentation*. Standard Performance Evaluation Corporation, release 1.01 edition, 2001.

[15] M. Süsskraut, S. Weigert, U. Schiffel, T. Knauth, M. Nowack, D. B. Brum, and C. Fetzer. Speculation for Parallelizing Runtime Checks. In *Proceedings of the 11th International Symposium on Stabilization, Safety, and Security of Distributed Systems*, pages 698–710, 2009.

[16] M. T. Vechev, E. Yahav, and D. F. Bacon. Correctness-Preserving Derivation of Concurrent Garbage Collection Algorithms. In *ACM Conference on Programming Languages Design and Implementation*, pages 341–353, 2006.

[17] S. Wallace and K. Hazelwood. SuperPin: Parallelizing Dynamic Instrumentation for Real-Time Performance. In *International Symposium on Code Generation and Optimization*, pages 209–220, 2007.

[18] A. Welc, S. Jagannathan, and A. Hosking. Safe futures for Java. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 439–453, 2005.

[19] T. Yuasa. Real-time garbage collection on general-purpose machines. *Journal of Systems and Software*, 11(3):181–198, 1990.