Artifact: The Anchor Verifier for Blocking and Non-Blocking Concurrent Software

Cormac Flanagan (UC, Santa Cruz)

Stephen Freund (Williams College)

Contents

Getting Started

Downloading

The artifact for our paper is packaged as a virtual machine available at the link provided.

Overview

We created and tested this Ubuntu 16 VM in VMWare Fusion 11.5.5. We have also run it in VirtualBox 6.1.12.

The VM contains the software infrastructure to use Anchor. The username and password to log in are both `artifact`. Terminal windows and FireFox can be opened from the toolbar on the left.

Once logged in, this file can be found in `~/Synchronicity/workspace/artifact/artifact.html`. That version includes clickable links to a number of other files, so we suggest opening it in Firefox as you proceed through the steps below.

Below you will find Quick Start instructions to run Anchor on some small prorgams, as well as instructions to run the benchmark experiments reported in the paper.

Machine Requirements

We used a real machine with a 3.2 GHz 6-Core Intel Core i7 process and 64 GB of memory running MacOS 10.15 to generate the results in the paper. Any reasonably-sized computer or virtual machine should be able to run all experiments, with perhaps some variability in timing results. When using our VM, here are two general configurations and what to expect:

- **Preferred:** The VM we provide runs Ubuntu 16 and is initially set to use 8GB of memory and 2 CPUs. That size should be suitable for running Anchor on all programs.
- **Minimal:** With 4GB and 1 CPU, most tests and benchmarks will run, but you may encounter failures due to memory depletion and/or see some slowdowns.

Relevant Papers and Books

- Anchor (OOPSLA 2020): Minor revision to fix typos in examples. ~/Synchronicity/workspace/artifact/docs/anchor-paper-revised.pdf (docs/anchor-paper-revised.pdf)
- VerifiedFT (PPoPP 2018): https://dl.acm.org/doi/10.1145/3178487.3178514 (https://dl.acm.org/doi/10.1145/3178487.3178514)
- CIVL (Concur 2015): https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/paper-12.pdf (https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/paper-12.pdf)
- Maurice Herlihy and Nir Shavit, *The Art of Multiprocessor Programming*, Morgan Kaufmann, 2012.

Quick Start

0. Launch a fresh terminal window.

1. Set up environment variables and tools.

```
> cd Synchronicity/workspace/Synchronicity
> source msetup
Anchor Configuration
--------------------
ANCHOR_HOME = /home/artifact/Synchronicity/workspace/Synchronicity

Commands
--------
anchor             -> /home/artifact/Synchronicity/workspace/Synchronicity/commands/anchor
anchor-benchmarks  -> /home/artifact/Synchronicity/workspace/Synchronicity/commands/anchor-benchmarks
compile-benchmarks -> /home/artifact/Synchronicity/workspace/Synchronicity/commands/compile-benchmarks
```

2. Use anchor to verify a small file to sanity check the configuration.

```
> cd ~/Synchronicity/workspace/artifact
> cat tests/ok1.anchor

class Ok {
  int y moves_as guarded_by this;
  public void inc()  {
    synchronized(this) {
      this.y = this.y + 1;
    }
  }
}

> anchor -quiet tests/ok1.anchor

Processing tests/ok1.anchor...

Verified!  (1.42 seconds)
```

The command line flag `-quiet` turns off most anchor logging messages. You may run without that flag to see more details about how anchor operates, although the details will not be relevant to this tutorial.

3. Use anchor on a second file that contains a reduction error:

```
> cat tests/bad1.anchor

class Bad {
  volatile int y moves_as isLocal(this) ? B : N;
  public void inc()  {
    this.y = this.y + 1;  // error: two non-mover accesses
  }
}

> anchor -quiet tests/bad1.anchor

Processing tests/bad1.anchor...

(6.5): Reduction failure

               this.y = this.y + 1;  // error: two non-mover accesses
               ^

  Trace:
     (6)     [N] tmp2 := this.y;                   {!isLocal(this, tid)  ==>  N}
     (6)         tmp3 = 1;
     (6)         tmp1 = tmp2 + tmp3;
     (6)     [N] this.y := tmp1;                   {!isLocal(this, tid)  ==>  N}

1 Errors.  (1.37 seconds)
```

The trace shows the execution path through the method body leading to the error in a lower-level translation of the original anchor source. Each step of the trace containing a synchronization or memory operation is structured as follows:

Line	Commutitivity	Operation	Commutivitity Computation

Line	Commutitivity	Operation	Commutivitity Computation
(6)	[N]	`tmp2 := this.y;`	`!isLocal(this, tid) ==> N`
(6)		`tmp3 = 1;`	
(6)		`tmp1 = tmp2 + tmp3;`	
(6)	[N]	`this.y := tmp1;`	`!isLocal(this, tid) ==> N`

These lines reflect a possible execution that leads to a reduction failure, that is, a `yield`-free sequence whose operations do not match the reducible pattern $R^{*}[N]L^{*}$. For the first and fourth steps, `this` is not thread-local, so the synchronization specification

```
isLocal(this) ? B : N
```

indicates that the accesses will be non-movers. For the second and third, only local variables are accessed, so both steps are trivially both-movers. In trivial cases like this, we omit the commutivity from the trace for improved readability.

4. To view the heap snaphots corresponding to the error trace in the previous example, run anchor again with the `-snapshots` flag:

```
> anchor -quiet -snapshots tests/bad1.anchor
...
1 Errors.  (1.38 seconds)
> firefox log/errors-frames.html
```

The last line uses FireFox to open the local webpage `log/errors-frames.html`, which contains the error trace. Clicking on a memory or synchronization operation will bring up the heap snapshot of the pre-state for that line of code. All local variables appear in the frame titled `LOCALS`. Object references link to object records with names like `Bad.0`. The `LOCALS` frame is typically in the lower left of the snapshot image.

In this case, the only relevent item to note in the heap snapshots is that the `this` object is in the *Shared* state, and thus not local to the current thread. There are several special local variables in the diagram:

- `tid`: the thread id for the currently executing thread.
- `_pc`: the phase of the current thread, as defined in the thread-modular semantics. Its value is either *Pre* or *Post*.

There are also several special fields in each object:

- `_lock`: stores the id of the thread holding the lock, or -1 if it is not held.
- `State`: one of `Fresh`, `Local(n)`, or `Shared`. These represent unallocated objects, objects local to thread *n*, and objects that are potential shared among multiple threads, respectively.

Note: Anchor shows the full counterexample reported by the underlying Boogie verifier. This may include many extraneous objects, depending on how exactly the SMT solver discovered the error. Our tool could filter out the irrelevent parts, but we have chosen to left them in to aid in debugging our verifier.

Step-By-Step Instructions

Claims Validated With Artifact

1. Anchor supports automatically verifies that programs are race-free, do not go wrong (due to assertaion failures), respect their synchronization disciplines, and are P/C equivalent.
2. Anchor supports synchronization specifications that can be checked for self-consistency properties independent of implementation code.
3. Anchor provides diagnostic error messages to the programmer to support finding and fixing concurrent errors.
4. When provided, Anchor verifies functional specifications (written as post conditions) for public methods.
5. Anchor can verify complex algorithms not easily handled by the most closely related existing techniques (eg: CIVL).
6. Anchor provides a better intuitive model for which code fragments are reducible.
7. Anchor has the specification and verification time overheads reported in the paper.
8. Verified Anchor programs can be compiled and executed.

We believe this list covers the major claims from the paper.

Claims 1 and 2: Demonstrated by Verifying Examples in the Paper and Others

Below we list the major examples used in the body of the paper, and several other small, illustrative programs. Collectively, they demonstrate Anchor's ability to verify the correctness criteria listed in Claim 1 and 2 by showing both positive and negative examples of the correctness properties listed.

For each program, we have a link to the source code, the comment line to run that code, and the expected result. All code for these examples is in the `workspace/artifact/papers` directory.

Follow the set up steps as above before running these examples if you are in a new shell, and also `cd` into the `artifact` directory:

```
> cd ~/Synchronicity/workspace/Synchronicity
> source msetup
...
> cd ~/Synchronicity/workspace/artifact
```

- **Paper Figure 1:** AtomicInt.anchor (paper/AtomicInt.anchor)

  ```
  > anchor -quiet paper/AtomicInt.anchor
  ...
  Verified!  (1.69 seconds)
  ```

- **Paper Figure 3:** LockBasedStack.anchor (paper/LockBasedStack.anchor)

  ```
  > anchor -quiet paper/LockBasedStack.anchor
  ...
  Verified!  (2.40 seconds)
  ```

- **Paper Figure 3:** LockFreeStack.anchor (paper/LockFreeStack.anchor)

  ```
  > anchor -quiet paper/LockFreeStack.anchor
  ...
  Verified!  (2.90 seconds)
  ```

- **Validity Error:** ValidityError.anchor (paper/ValidityError.anchor)

  A more interesting example of the validity error appearing in Figure 1. This shows that Anchor verifies synchronizations disciplines independent of implementation code, and that it rejects programs for which the synchronization disciplines do not imply race freedom. (The validity requirements must be satisfied to guarantee a spec rules out races.)

```
> cat paper/ValidityError.anchor
class BadDiscipline {
    volatile int value moves_as isRead() ? B : N;
}
> anchor -quiet paper/ValidityError.anchor
Processing paper/ValidityError.anchor...

(2.5): BadDiscipline.value failed Read-Write Right-Mover Check

    volatile int value moves_as isRead() ? B : N;
    ^
    Thread t    : _ := x.value;  [(isRead)  ==>  B]
    Thread u    : x.value := 6;  [!isRead  ==>  N]

(2.5): BadDiscipline.value failed Read-Write Left-Mover Check

    volatile int value moves_as isRead() ? B : N;
    ^
    Thread t    : _ := x.value;  [(isRead)  ==>  B]
    Thread u    : x.value := 5;  [!isRead  ==>  N]

...
3 Errors.  (1.25 seconds)
```

These two errors indicate that the `[Valid Read-R]` and `[Valid Read-L]` validity requirements in Definition 5.1 are not met by the synchronizaiton discipline for the `value` field. Specifically, thread *t* may read the `value` field of an object while another thread *u* can change it. Note that this error is reported using *only* the synchronization specification.

The third error, which we elided above, is due to a minor detail relating to the implicit `yields_as` annotation inserted by the verifer for field `value`. Specs failing the validity checks will never conform to the basic requirements for `yeilds_as` annotations outlined in Section 8 of the paper. (In reality, we should probably suppress this warning since the root cause is validity errors.)

Claim 3: Demonstrated by Verifying Examples in the Paper

We show below the three `buggy` code examples from Section 2 of the paper to illustrate how Anchor reports errors in a way that facilitates debugging and fixing synchronization errors.

- **Paper Section 2: Anchor Error Messages: buggy() Version 1 (paper/ErrorMessage1.anchor)**

```
> anchor -quiet paper/ErrorMessage1.anchor
...
(40.5): Reduction failure

   assert this.head.item == 10;
   ^

 Trace:
    (38)        tmp10 = 10;
    (38)        inlined this.push(tmp10) ;
    (38)          item_2 = tmp10;
    (38)          this_2 = this;
    (16)    [R]   acquire(this_2);
    (17)          node_2 = new Node();
    (17)    [B]   tmp1_2 := this_2.head;                {!isLocal(this_2, tid) && (holds(this_2, tid))  ==>  B}
    (17)          inlined node_2.init(item_2,tmp1_2) ;
    (17)            item_2_3 = item_2;
    (17)            next_2_3 = tmp1_2;
    (17)            this_2_3 = node_2;
    (6)     [B]     this_2_3.item := item_2_3;          {(isLocal(this_2_3, tid)) &&
                                                         (isLocal(this_2_3, tid))  ==>  B}
    (7)     [B]     this_2_3.next := next_2_3;          {(isLocal(this_2_3, tid)) &&
                                                         (isLocal(this_2_3, tid))  ==>  B}
    (5)             break exit_2_3;
    (18)    [B]   this_2.head := node_2;                {!isLocal(this_2, tid) &&
                                                         (holds(this_2, tid))  ==>  B}
    (19)    [L]   release(this_2);
    (40)    [E] tmp13 := this.head;                     {!isLocal(this, tid) && !holds(this, tid)  ==>  E}

1 Errors.  (2.73 seconds)
```

The most closely related contemporary tool (eg: CIVL) may show a counterexample trace, but does not provide any detail as to *why* the reduction error occurred, merely that it did.

- **Paper Section 2: Anchor Error Messages: buggy() Version 2 (paper/ErrorMessage2.anchor)**

```
> anchor -quiet paper/ErrorMessage2.anchor
...
(39.5): Reduction failure

   acquire(this);
   ^
 Trace:
    (38)        tmp10 = 10;
    (38)        inlined this.push(tmp10) ;
    (38)          item_2 = tmp10;
    (38)          this_2 = this;
    (16)    [R]   acquire(this_2);
    (17)          node_2 = new Node();
    (17)    [B]   tmp1_2 := this_2.head;                {!isLocal(this_2, tid) && (holds(this_2, tid))  ==>  B}
    (17)          inlined node_2.init(item_2,tmp1_2) ;
    (17)            item_2_3 = item_2;
    (17)            next_2_3 = tmp1_2;
    (17)            this_2_3 = node_2;
    (6)     [B]     this_2_3.item := item_2_3;          {(isLocal(this_2_3, tid)) &&
                                                         (isLocal(this_2_3, tid))  ==>  B}
    (7)     [B]     this_2_3.next := next_2_3;          {(isLocal(this_2_3, tid)) &&
                                                         (isLocal(this_2_3, tid))  ==>  B}
    (5)             break exit_2_3;
    (18)    [B]   this_2.head := node_2;                {!isLocal(this_2, tid) && (holds(this_2, tid))  ==>  B}
    (19)    [L]   release(this_2);
    (39)    [R] acquire(this);

1 Errors.  (2.60 seconds)
```

- **Paper Section 2: Anchor Error Messages: buggy() Version 3 (paper/ErrorMessage3.anchor)**

In addition to illustrating more of Anchor's diagnostic error message features, this example demonstrates that Anchor rejects programs with `assert` statements that may fail.

In this case, we use the `-snapshots` flag to generate the associated heaps.

```
> anchor -snapshots -quiet paper/ErrorMessage3.anchor
...
(41.5): This assertion may not hold.

   assert this.head.item == 10;
   ^
...
1 Errors.  (2.77 seconds)
> firefox log/errors-frames.html
```

In the errors web page, you can see the execution trace leading to the assertion failure.

If you click on the hyperlink for `assert tmp11` in the error trace, you will see the heap snapshot at the point of failure. Details may vary depending on the exact counterexample produced by the underlying SMT solver, but you should be able to see that `this` in `LOCALS` refers to a stack whose `head.item` value is not 10.

If you click on the hyperlink for the yield in the error trace, you will see the heap before and after the yield point. Differences are marked in red, and you should see that `this.head` changed from a node with an `item` value of 10 to one with a different value in `item`.

Other SMT-based tools in the domain of concurrent program verification do not show a visualization of the concrete counterexample.

Claim 4: Demonstrated by Verifying Examples in the Paper and Others

We provide several examples with method specifications to demonstrate that Anchor will also verify post conditions for public methods.

- **Paper Figure 8: LockBasedStackWithSpec.anchor (paper/LockBasedStackWithSpec.anchor)**

  When verifying method post conditions, we employ the anchor flag `-verify=MethodSpecs`.

  ```
  > anchor -verify=MethodSpecs -quiet paper/LockBasedStackWithSpec.anchor
  ...
  Verified!  (2.90 seconds)
  ```

  To demonstrate a failure when verifying method specifications, you can comment out line 20 of that file (`this.head = node;`) and run again. That line is marked in the file for your convenience.

  ```
  > anchor -verify=MethodSpecs -quiet paper/LockBasedStackWithSpec.anchor
  ...
  (17.30): Method returns before completing actions in spec
  ...
  ```

  Caveat: Specification failure error messages contain some rough diagnostic info that we find useful for debugging our verifier, but that info is not necessarily intended for the end programmer.

- **LockFreeStackWithSpec.anchor (paper/LockFreeStackWithSpec.anchor)**

  ```
  > anchor -quiet -verify=MethodSpecs paper/LockFreeStackWithSpec.anchor
  Processing paper/LockFreeStackWithSpec.anchor...
  Verified!  (3.03 seconds)
  ```

- **FineGrainedListWithSpec.anchor (paper/FineGrainedListWithSpec.anchor)**

```
> anchor -quiet -verify=MethodSpecs paper/FineGrainedListWithSpec.anchor
Processing paper/FineGrainedListWithSpec.anchor...
Verified!  (3.15 seconds)
```

Claims 5, 6, 7: Demonstrated by Verifying the Examples in Table 1

1. Follow the set up steps as above before running the benchmarks if you are in a new shell:

```
> cd ~/Synchronicity/workspace/Synchronicity
> source msetup
...
```

2. The source files for the examples in Table 1 are stored in `~/Synchronicity/workspace/Synchronicity/benchmarks/anchor`.
   To verify each example once, use the following command. This should complete in about 5-8 minutes.

```
> cd ~/Synchronicity/workspace/Synchronicity
> anchor-benchmarks
Processing benchmarks/anchor/stack-coarse.anchor...
Verified!  (3.65 seconds)
Processing benchmarks/anchor/stack-lock-free.anchor...
Verified!  (2.99 seconds)
Processing benchmarks/anchor/list-coarse.anchor...
Verified!  (4.18 seconds)
Processing benchmarks/anchor/list-fine.anchor...
Verified!  (2.86 seconds)
Processing benchmarks/anchor/list-optimistic.anchor...
Verified!  (3.70 seconds)
Processing benchmarks/anchor/list-lazy.anchor...
Verified!  (4.06 seconds)
Processing benchmarks/anchor/list-lock-free.anchor...
Verified!  (24.36 seconds)
Processing benchmarks/anchor/queue-coarse.anchor...
Verified!  (4.35 seconds)
Processing benchmarks/anchor/queue-lock-free.anchor...
Verified!  (5.12 seconds)
Processing benchmarks/anchor/queue-bounded.anchor...
Verified!  (12.84 seconds)
Processing benchmarks/anchor/hashset-coarse.anchor...
Verified!  (14.30 seconds)
Processing benchmarks/anchor/hashset-striped.anchor...
Verified!  (188.62 seconds)
Processing benchmarks/anchor/fasttrack.anchor...
Verified!  (22.21 seconds)
```

Caveat: The time required to verify a program depends on many factors, including the non-determinism inherent in the underlying Z3 SMT solver. We have found that moving from one computer to another or simply repeating experiments under slightly different conditions can lead to fairly sizable time differences, particularly for `queue-bounded.anchor` and `hashset-striped.anchor`. Thus, while the numbers above are typical for the VM on our hosts, they may not be entirely congruent with what you observe.

3. To verify each multiple times and report both annotation and timing statistics, use a command like the following. The command will print out a latex table that you may use to compare your timing results to those in the paper.

```
> cd ~/Synchronicity/workspace/Synchronicity
> # Runs each 10 times, reporting average time.  Hopefully < 1 hour.
> anchor-benchmarks -n=10
Stacks
  Coarse...
    Running with command: ./commands/time-anchor -quiet -B=-timeLimit:600  benchmarks/anchor/stack-coarse.anchor
      Warmup Run 1: 2.59 seconds
      ...
      Run 1: 2.98 seconds
      Run 2: 2.97 seconds
      Run 3: 3.03 seconds
      ...
      Run 10: 3.17 seconds
  Lock-free...
    ...
Linked Lists
  ...
...
> pdflatex data.tex
> gnome-open data.pdf
```

If there are any errors on a benchmark, the time will be reported as 0 and a message will tell you how to re-run the benchmark to observe the problem – we can help diagnose any errors from the output of that run.

We provide here logs and tables for several test machines to illustrate the expected output format and timings on
different systems. As noted above, you may see some differences for several test programs.
These files are stored in `~/Synchronicity/workspace/artifact/`.

- Virtual Machine, 2 Cores, 8 GB RAM, Ubuntu 16. Hosted on Mac Mini (next config).
  - Data in `data/vm-osx/`: Log file (data/vm-osx/log.txt) and PDF Table (data/vm-osx/data.pdf)

- Apple Mac Mini, 3.2 GHz 6-Core Intel Core i7, 64 GB RAM, OSX 10.15.
  - Data in `data/mac-osx/`: Log file (data/mac-osx/log.txt) and PDF Table (data/mac-osx/data.pdf)

Claims 4, 5, 6, 7: Demonstrated by the FastTrack Case Study

We use the following measurements as our basis of comparision between the CIVL and Anchor implementations of the FastTrack dynamic race detector core algorithm. This case study further demonstrates Anchor's ability to verify functional specifications, and it also shows that Anchor can readily verify complex synchronization and algorithms and provides an intuitive programming model by enabling to easily and succinctly encode both the FastTrack algorithm and specification. We also show how to reproduce the timing measures reported for the variants of FastTrack.

Note: This section involves verifying full functional specifications of the FastTrack code, not just P/C equivalence.

1. **Code and Spec Size.** We include five variants of FastTrack.

   Two are implementations in CIVL:

   - ~/Synchronicity/workspace/artifact/verified-ft/verified-ft-code-only.bpl (verified-ft/verified-ft-code-only.bpl): the previously published CIVL implementation with no specs and no test harness.
   - ~/Synchronicity/workspace/artifact/verified-ft/verified-ft-no-client.bpl (verified-ft/verified-ft-no-client.bpl): the previously published CIVL implementation with full specs and no test harness.

   Three are implementations in Anchor:

   - ~/Synchronicity/workspace/artifact/verified-ft/fasttrack-original.anchor (verified-ft/fasttrack-original.anchor): our Anchor implementation with no specs.
   - ~/Synchronicity/workspace/Synchronicity/benchmarks/anchor/fasttrack.anchor (../Synchronicity/benchmarks/anchor/fasttrack.anchor): our Anchor implementation with the specs necessary to prove P/C equivalence. (A more readable version is fasttrack-with-macros.anchor (../Synchronicity/benchmarks/verified-ft-

src/fasttrack-with-macros.anchor). That version uses C macros for common operations in the spec.)

- ~/Synchronicity/workspace/Synchronicity/benchmarks/specs/fasttrack.anchor (../Synchronicity/benchmarks/specs/fasttrack.anchor): our Anchor implementation with full functional specifications matching those in the CIVL code. (A more readable version is fasttrack-specs-with-macros.anchor (../Synchronicity/benchmarks/verified-ft-src/fasttrack-specs-with-macros.anchor). That version uses C macros for common operations in the spec.)

Here are their sizes (lines of source code with empty lines and comments removed):

```
> cd ~/Synchronicity/workspace/artifact/verified-vt
> ./lines.sh
File                                                                          LOC
==========================================================================    ===
~/Synchronicity/workspace/artifact/verified-ft/verified-ft-code-only.bpl      260
~/Synchronicity/workspace/artifact/verified-ft/verified-ft-no-client.bpl      789
--------------------------------------------------------------------------
~/Synchronicity/workspace/artifact/verified-ft/fasttrack-original.anchor      217
~/Synchronicity/workspace/Synchronicity/benchmarks/anchor/fasttrack.anchor    259
~/Synchronicity/workspace/Synchronicity/benchmarks/specs/fasttrack.anchor     385
```

From these, the sizes of the specifications are computed as follows:

Spec	**Size (LOC)**	**Size (%)**
CIVL spec	789 – 260 = 529 lines	529/260 = 203%
Anchor spec for P/C equivalence	259 – 217 = 42 lines	42/217 = 19%
Anchor spec for functional corretness	385 – 217 = 168 lines	168/217 = 77%

Our claims related to handling such complex code more readily and intuitively are based on the dramatically reduced specification requirements, from 529 lines to 168 lines.

2. **Verification Time.** We measured FastTrack verification under CIVL and Anchor with the following commands. Timings will differ from those reported in the paper under the VM and due to improvements in the Anchor verifier made after the initial paper submission.

Single Runs of CIVL and Anchor:

```
> cd ~/Synchronicity/workspace/artifact/verified-ft
> time bpl verified-ft-no-client.bpl
Boogie program verifier version 2.4.1.10503, Copyright (c) 2003-2014, Microsoft.

Boogie program verifier finished with 522 verified, 0 errors

real    0m9.786s
user    0m10.629s
sys 0m0.370s
```

The real running time for CIVL is 9.876 seconds.

```
> cd ~/Synchronicity/workspace/Synchronicity
> anchor -quiet -verify=MethodSpecs benchmarks/specs/fasttrack.anchor
...
Verified (51.49 seconds)
```

The real running time for Anchor is 51.49 seconds.

Average of Multiple Runs of CIVL and Anchor:

```
> cd ~/Synchronicity/workspace/artifact/verified-ft
> time -p for i in {1..10}; do ../../Synchronicity/bpl -nologo verified-ft-no-client.bpl; done

Boogie program verifier finished with 522 verified, 0 errors

Boogie program verifier finished with 522 verified, 0 errors

...

Boogie program verifier finished with 522 verified, 0 errors
real 101.86
user 104.34
sys 4.09
```

So the average real time for CIVL is 101.86/10 = 10.2 seconds.

For Anchor:

```
> cd ~/Synchronicity/workspace/Synchronicity
> time -p for i in {1..10}; do anchor -quiet -verify=MethodSpecs benchmarks/specs/fasttrack.anchor; done
Processing benchmarks/specs/fasttrack.anchor...
Verified!  (50.29 seconds)
Processing benchmarks/specs/fasttrack.anchor...
Verified!  (48.96 seconds)
Processing benchmarks/specs/fasttrack.anchor...
Verified!  (50.33 seconds)
Processing benchmarks/specs/fasttrack.anchor...
Verified!  (51.54 seconds)
Processing benchmarks/specs/fasttrack.anchor...
Verified!  (50.51 seconds)
Processing benchmarks/specs/fasttrack.anchor...
Verified!  (49.40 seconds)
Processing benchmarks/specs/fasttrack.anchor...
Verified!  (50.94 seconds)
Processing benchmarks/specs/fasttrack.anchor...
Verified!  (54.09 seconds)
Processing benchmarks/specs/fasttrack.anchor...
Verified!  (48.51 seconds)
Processing benchmarks/specs/fasttrack.anchor...
Verified!  (48.75 seconds)
real 508.10
user 620.35
sys 28.41
```

So the average real time for Anchor is 508.10/10 = 50.8 seconds.

This experiment can also yield fairly different timing results:

- Virtual Machine, 2 Cores, 8 GB RAM, Ubuntu 16. Hosted on Mac Mini (next config).
  - CIVL: ~10 seconds
  - Anchor: ~51 seconds
- Apple Mac Mini, 3.2 GHz 6-Core Intel Core i7, 64 GB RAM, OSX 10.15.
  - CIVL: ~10 seconds
  - Anchor: ~171 seconds

Claim 8: Demonstrated by Compiling and Testing the Examples in Table 1

To compile and test the examples in Table 1, you can use the following command:

```
> cd ~/Synchronicity/workspace/Synchronicity
> compile-benchmarks -quiet
### stack-coarse

Generating Jar File benchmarks/jars/stack-coarse.jar

Running Unit Tests
JUnit version 4.12
.sequential push and pop
.parallel both
.parallel push
.parallel pop

Time: 0.007

OK (4 tests)


### stack-lock-free
...
```

All examples should end with `OK`. The compiler generates Java code, which is in `benchmarks/java`. That code is then compiled and bundled into jar files in `benchmarks/jars`. The unit tests are in `benchmarks/unit-tests`. They are based on the tests that come with the source from the Herlihy and Shavit book.

Building and Unit Tests

The Anchor source can be recompiled as follows:

```
> cd ~/Synchronicity/workspace/Synchronicity
> make
mkdir -p bin
javac -d bin -classpath ".:tools/*" `find src -name "*.java"`
fsc -d bin -classpath ".:tools/*" `find src -name "*.java"` `find src -name "*.scala"`
> # run unit tests in ./tests directory (15-30 minutes)
> make tests
...
Run starting. Expected test count is: 196
SpecTests:
Spec Tests
- Example2.anchor should succeed
- a.anchor should succeed
- aba.anchor should fail with 1 errors
- annot-1.sink should fail with 1 errors
- array-bad-1.sink should fail with 14 errors
...
```

While not part of the submitted artifact, if you wish to install Anchor on a different computer, please follow the installation instructions (install.html). We believe them to be relatively portable to different platforms, but your mileage may vary…

Known Issues And Caveats

- The Z3 solver's proof search is non-deterministic. As such, switching machines, repeating queries, making small changes in the input, etc. can all have an impact on performance. This impacts some test inputs more than others, and you may see observe different timing results for some programs than we observed on our test machine. In the extreme, this can lead to non-termination for programs we were able to verify. We don't anticipate that happening but can provide some guidance to work around that limitation.
- Anchor shows the full counterexample reported by the underlying Boogie verifier. This may include many extraneous objects, depending on how exactly the SMT solver discovered the error. Our tool could filter out the irrelevent parts, but we have chosen to left them in to aid in debugging our verifier.
- Anchor only parses a subset of the format for counterexamples produced by Boogie/Z3. While we have not encountered problems resulting from this in the examples presented here, if Anchor reports warnings about the Boogie counterexample, it is due to this limitation and parts of the counterexample will not be presented in the heap snapshot.