QuickerCheck

Robert Krook
krookr@chalmers.se
Chalmers University of Technology
Gothenburg, Sweden

Nicholas Smallbone
nicsma@chalmers.se
Chalmers University of Technology
Gothenburg, Sweden

Koen Claessen
koen@chalmers.se
Chalmers University of Technology
Gothenburg, Sweden

ABSTRACT

Finding faults in complex software is not trivial, and writing unit tests that properly test a complex software will take an incredible amount of time. If you have a specification of how the software should behave, you can instead turn that specification into properties and generate random test data, and test that the specification is not violated for that data. Instead of spending valuable human-hours writing unit tests, we can now spend cheaper computer-hours running randomly generated tests. However, it might still take a non-trivial amount of time to run a meaningful number of tests.

In this paper, we describe a new parallel run-time for QuickCheck, a Haskell EDSL for specifying and testing properties of programs. The new run-time evaluates individual properties in parallel, distributing all tests over the available cores. When a counterexample is found, it will be shrunk in parallel.

The results indicate that there is a lot to gain in terms of performance when more cores are added and that the penalty of the new implementation when only one core is used, compared to the existing QuickCheck implementation, is low. When more cores are used, the new implementation can run many more tests in the same amount of time, ultimately finding bugs faster.

CCS CONCEPTS

• **Computing methodologies** → **Concurrent algorithms**; **Shared memory algorithms**; • **Theory of computation** → **Shared memory algorithms**; • **Software and its engineering** → **Software testing and debugging**.

KEYWORDS

property-based testing, quickcheck, testing, parallel functional programming, haskell

ACM Reference Format:
Robert Krook, Nicholas Smallbone, and Koen Claessen. 2018. QuickerCheck. In *Woodstock '18: ACM Symposium on Neural Gaze Detection, June 03–05, 2018, Woodstock, NY*. ACM, New York, NY, USA, 9 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

As a software system gets larger and larger, properly testing that system becomes an increasingly heavy burden. Writing unit tests for large systems made up of many moving parts is far from trivial, and writing enough tests to properly cover the functionality of a system might be a Herculean task on its own.

Thankfully, there are alternatives to unit testing that scale better. Rather than trying to come up with test cases manually, we can let our computer generate them for us. Historically, Fuzzing [5] has been a popular choice for testing software that takes some form of structured data as input. When you "fuzz" software, you generate arbitrary input for it and inspect if it crashes or hangs.

Fuzzing is an instantiation of the more general paradigm known as *Property-based Testing* (PBT). Rather than generating arbitrary input and monitoring the software for crashes, PBT may generate richer inputs and monitor the software not just for crashes, but for the expected behavior. PBT can be used to test whether some software behaves according to its specification or not.

QuickCheck [2] (QC) is a domain-specific language embedded in Haskell, that lets a developer not just specify the properties of the software they are testing, but also how to generate inputs and shrink counterexamples. Properties are written as ordinary Haskell functions, which may be pure or perform side effects. QC will take such a function and generate arbitrary input for it a configurable amount of times, and check if the property holds.

While this sounds simple, QC has been proven useful at testing complex systems such as compilers and large production databases [3]. A disadvantage of testing complex systems is that the properties might take a non-trivial amount of time to execute. Whereas a simple, pure property on e.g. lists might finish testing before you have time to blink, testing something like a compiler will require more time. Aside from the generation of programs being more complex, testing the property requires the generated program to be compiled and executed, which may be slow. Furthermore, if testing properties is done automatically as part of a test script in production, there is a limit to the amount of time allocated for this. If the tests take a long time to execute, there might not be enough time to run a meaningful number of tests.

This begs the question; can we test a property in parallel? Modern machines have multiple cores to the extent that it is difficult to get your hands on a single-core machine. We want to make use of all of these cores to run our tests in parallel. Pure properties are immediately embarrassingly parallel, while effectful properties can often be modified to be embarrassingly parallel. This will allow us to execute more tests in the same time period, hopefully shortening the time required to find a bug.

In this paper, we present a new implementation of QC that has support for running all tests of a property in parallel. Furthermore, if a counterexample is found, parallel shrinking can be used to find a smaller counterexample quicker. While tests themselves may be embarrassingly parallel, the internal test driver of QC is not. This makes the implementation non-trivial as there are choices to make.

Conference acronym 'XX, June 03–05, 2018, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

We try to motivate our implementation choices by both discussing them and evaluating them.

2 QUICKERCHECK

We present QuickerCheck via two examples. We point out that the QC EDSL for writing generators, shrinkers, and properties (see section 3) remains unchanged, and only the internal evaluation of a property is modified.

System F. System F is a variant of the simply typed lambda calculus that can quantify a program over polymorphic types. A property of system F is that if a term is of type `T`, then the same term after reducing one step should still be of type `T`. We state this in a property

```
prop_preservation :: Expr -> Property
prop_preservation e = case reduce e of
  Just e' -> typeOf e == typeOf e'
  Nothing -> discard
```

If the term could not be reduced further we simply discard it, as it is not relevant for the property we are specifying. As this is a pure Haskell function, this is an embarrassingly parallel function, and can immediately be executed in parallel. To evaluate the property in parallel, aside from compiling it with the `-threaded` flag, it is enough to invoke `quickCheckPar` instead of `quickCheck`, and pass in the `-N` option to the runtime system. The output (assuming all tests passed) is

```
> quickCheckPar prop_preservation
+++ OK! Passed 100 tests.
  tester 0: 50
  tester 1: 50
```

What is not visible in the output is that, if the command was executed on a machine with multiple cores, the 100 tests were executed faster.

Compiler testing. A function that is not necessarily embarrassingly parallel is one that is effectful. To test a compiler it is necessary to perform IO actions, such as invoking the compiler under test or executing the compiled binary. Testing compilers is non-trivial, but a well-studied approach is to define metamorphic relations, where known changes in the input should have a predictable change in the output. That is if we want to test a function `f : a -> b`, we can modify the input `a` with the function `trans : a -> a` and observe the output we would have gotten had we instead modified the original output with `transOutput : b -> b`. Stating this as a property

```
prop :: a -> Bool
prop x = transOutput (f x) == f (trans x)
```

We can specialize the property to test a compiler using some metamorphic relation described by `trans` and `transOutput`, and get the property below

```
prop_diff :: Program -> Property
prop_diff program = monadicIO $ do
  let pi = "p.qclang"
      qi = "q.qclang"
  run $ writeFile pi (render program)
  run $ writeFile qi (render $ trans program)
  output1 <- run $ compileAndRun pi
  output2 <- run $ compileAndRun qi
  mapM removeFile [ pi
                  , qi
                  , "p.exe"
                  , "q.exe"]
  assert (transOutput output1 == output2)
```

The property executes both the original and modified program after having first written them to the file system. The file system is cleaned up, after which the outputs are compared. The output of the unmodified program is modified to reflect the change described by the metamorphic relation.

The property above, while innocent looking, is not embarrassingly parallel. If multiple instances of this property execute in parallel, they implicitly share one resource, the file system. The effectful operations of using the file system leads to race conditions. There are different ways to modify the property such that there are no race conditions, of which one is described below

```
prop_diff :: IORef Int -> Program -> Property
prop_diff ng program = monadicIO $ do
  i <- run $ atomicModifyIORef ref $ \i -> (i+1, i)
  let pi = concat ["p", show i, ".qclang"]
      qi = concat ["q", show i, ".qclang"]
  run $ writeFile pi (render program)
  run $ writeFile qi (render $ trans program)
  output1 <- run $ compileAndRun pi
  output2 <- run $ compileAndRun qi
  mapM removeFile [ p1
                  , qi
                  , "p" <> show i <> ".exe"
                  , "q" <> show i <> ".exe"
                  ]
  assert (transOutput output1 == output2)
```

The property uses an IORef to get unique identifiers when accessing the file system. If we disregard other implicitly shared resources such as CPU caches, RAM, bandwidth, etc, this property is now embarrassingly parallel. An alternative patch is to always generate a fresh directory for each test and make that the working directory, but that requires that each thread be a *bound* thread (see section 4).

Shrinking. QC has support for automatically shrinking any found counterexample. Shrinking is a deterministic, iterative process where a counterexample is recursively shrunk until it can not be shrunk further. Conceptually, shrinking is the act of searching for the leftmost path down a rose tree, where each node is a mutation of its parent and every node on the path falsifies the property. QC never backtracks to see if it could have found a longer path down the tree but stops once a leaf has been encountered.

QuickerCheck has support for parallel shrinking. There are two variants, where the first returns the same result as the deterministic process described above. The candidate inputs are evaluated in parallel, and shrinking might yield a faster result. The second variant is more greedy and does not promise to produce a deterministic result. It returns as soon as any path down the tree has been found, and does not try to imitate the deterministic process. This might

yield different counterexamples for different runs but terminates faster.

3 BACKGROUND

QuickCheck. QC is an embedded domain-specific language for specifying and testing properties. Specifying a property is done by defining a Haskell function, and a widely recognized example property is the property that asserts that reversing a list twice returns the original list.

```
prop_reverse :: [Int] -> Bool
prop_reverse xs = xs == reverse (reverse xs)
```

QC can test this property for us by generating arbitrary inputs over and over, verifying that the function always returns `True`[1].

While QC natively has support for generating values of built-in types, the developer can define their own generators to generate arbitrary values of custom types. Such generators are described in the `Gen` monad. The `Gen` monad uses a random seed to make random choices, and a size parameter to limit how large the generated values are.

```
data Gen a = Gen { unGen :: QCGen -> Int -> a }
```

QC gives the developer access to many combinators which makes it very easy to implement generators. As an example, the `elements` combinator takes a list of values and returns a generator that selects one of them at random, with a uniform distribution.

```
arbitraryBool :: Gen Bool
arbitraryBool = elements [True, False]
```

The above generator is especially simple because we did not need to limit the size of the generated value. If we instead wish to generate e.g. an arbitrarily long list of booleans, we need to consider this. The `Int` parameter of the `Gen` monad can be accessed by generators through the `sized` combinator.

```
arbitraryBoolList :: Gen [Bool]
arbitraryBoolList = sized $ \size -> do
  s <- chooseInt (0, size)
  sequence $ replicate s arbitraryBool
```

The above generator will generate a number between 0 and the size, and generate an arbitrary list of that length. This can be further generalized to accept a generator of any type.

```
listOf :: Gen a -> Gen [a]
listOf g = sized $ \size -> do
  s <- chooseInt (0, size)
  sequence $ replicate s g
```

This is precisely how the `listOf` generator is implemented in QC.

The Test Loop. The internal test driver is a recursive function that maintains a state which contains e.g. the counts of how many tests were executed so far, how many were discarded, etc. It also holds the random seed used to generate the arbitrary input. A pseudo-function describing the general layout of the test loop is described below

[1]The authors would like to note here that this property holds for finite lists. Haskell's support for laziness allows us to invoke this property with an infinite list, at which point the program diverges.

```
test :: State -> Property -> IO Report
test st f | discardedTooMany st = giveUp st
          | ranEnoughTests st   = done st
          | otherwise = do
  let (s1,s2) = split (seed st)
      (n, d)  = (numSuccess st, numDiscarded st)
      size    = computeSize n d
  res <- unGen (unProperty f) s1 size
  case res of
    Just False -> failed st res
    Just True  -> test (st { seed = s2, numSuccess   = n + 1}) f
    Nothing    -> test (st { seed = s2, numDiscarded = d + 1}) f
```

The function checks if we discarded or executed enough tests. In that case, recursion is broken and a report is returned. Otherwise, the loop will split the random seed and compute the size for the next test case. The property is then evaluated, after which the result is inspected. There are three potential outcomes – either a counterexample was found, the property was satisfied, or the test case was discarded. Recursion is broken if a counterexample is found, and shrinking takes over.

Shrinking. QC can shrink counterexamples to find a smaller one. A developer can choose to implement the function `shrink :: a -> [a]` from the `Arbitrary` type class (An instance of the class `Arbitrary` specifies how to generate and shrink values). The function should take a value and return a list of mutants, where each mutant is smaller. As an example, consider the datatype for trees below

```
data Tree a = Leaf | Node (Tree a) a (Tree a)
```

If we have a value of type `Tree a`, where we know how to shrink values of type `a`, we can define a *shrinker* like this

```
shrink :: Tree a -> [Tree a]
shrink Leaf          = []
shrink (Node l a r) =
  concat [ [ Leaf, l, r]
         , [ Node l' a' r' | l' <- shrink (l,a,r)]
         ]
```

A `Leaf` can not be shrunk further, so an empty list of mutants is returned. A node, however, can be shrunk further. It is common to return the *greedy* mutants first, which are those mutants that discard large parts of the value. In this case, we can discard the entire tree, or we can choose to keep just one of the child trees. If none of these mutants falsify the properties, we keep the node but shrink the subterms.

QC will compute the mutants of a counterexample and traverse the list until a new counterexample is found or the list is empty. If a new counterexample is found, it will then recursively shrink that one. This is implemented by two mutually recursive functions `localMin` and `localMin'`.

```
localMin :: State -> Res -> [Res] -> IO (Int, Int, Int, Res)
localMin st res ts = do
  r <- tryEvaluate ts -- evaluate list to WHNF
  case r of
    Left err -> raiseException "exception generating shrinks"
    Right ts' -> localMin' st res ts'
```

```
localMin' :: State -> Res -> [Res] -> IO (Int, Int, Int, Res)
localMin' st res [] = doneShrinking st res
localMin' st res (t:ts) = do
  (r,rs) <- evaluateCandidate t
  if ok r == Just False
    -- found new counterexample, recurse with the new
    -- candidate `r` and mutants `rs`
    then localMin (updateAfterShrink st) r rs
    -- candidate did not falsify property, recurse with
    -- existing counterexample `res` and `ts`
    else localMin (updateAfterUnsuccessful st) res ts
```

Parallel Haskell. Draft note: this paragraph will (at least) cover

- Green threads in Haskell, bound vs unbound threads and the consequences of using either.
- MVars - shared, mutable variables.
- Asynchronous exceptions

4 DESIGN

To not modify what QC depends on, we choose to only use packages from `base`. We use the `Control.Concurrent` package to spawn green threads, whose implementation is described in [4]. While the default behavior is to spawn threads using `forkIO`, QuickerCheck can be configured to spawn bound threads using `forkOS` instead. Worker threads communicate as little as possible, but access shared resources through `MVars` from `Control.Concurrent.MVar`. To send each other signals, worker threads use asynchronous exceptions from `Control.Exception`.

Testing. To parallelize the testing loop, we run multiple instances of the sequential test loop simultaneously. Each such worker thread, which we will refer to as a tester, has its own instance of the state, which we might refer to as the *thread-local state*, that is fed into the recursive test loop. This state is updated independently of the state of the other testers, and will only reflect how many tests this specific tester has run.

This design sounds simple but introduces a problem that needs to be addressed. The function that computes the size imposes a sequential ordering to tests, as the size for one test depends on the outcome of previous tests. The sizes are derived from the number of tests that passed so far, and how many tests that were discarded since the last passing test case. If every tester computes the size using their own thread-local state, they are going to compute the same sizes! This is illustrated in figure 1b. To explore the full space of sizes, a tester may use either a size offset, as illustrated in figure 1c, or a size stride, as illustrated in figure 1d.

Communication between testers should be minimal. To avoid constant synchronization on how many tests are left to run, each tester is initially given an equal share of the expected number of tests to run. This number is decremented after every test, and only when a testers such counter reaches zero might it look at other testers. If *right-to-work-steal* is enabled, the tester will at this point see if any other tester has any allowance left, and may steal the right to run another test and compute a seed and size from its thread-local state. If this is not enabled, the tester will terminate. The total number of tests that can be discarded is handled in a similar manner.

With the `chatty` flag set to `True`, QC will print how many tests have been executed and discarded after *every* test. While helpful to illustrate progress, printing is a very expensive operation. If tests run very quickly, the cost of printing the current progress will dominate the total execution time. To improve performance, QuickerCheck should print the current progress less frequently.

Shrinking. The existing shrink loop continually evaluates the head of the candidate list until a new counterexample is found, at which point the loop recurses, or until the list is empty, at which point shrinking is terminated. The design of the new loop is very similar.

Rather than a single thread traversing the candidate list one element at a time, the parallel shrink loop spawns concurrent worker threads that cooperate and traverses the same list. If any of the concurrent workers find a new counterexample, they will update the shared list of candidates and signal to their sibling workers that they should stop evaluating their current candidate and instead pick a new one from the new list.

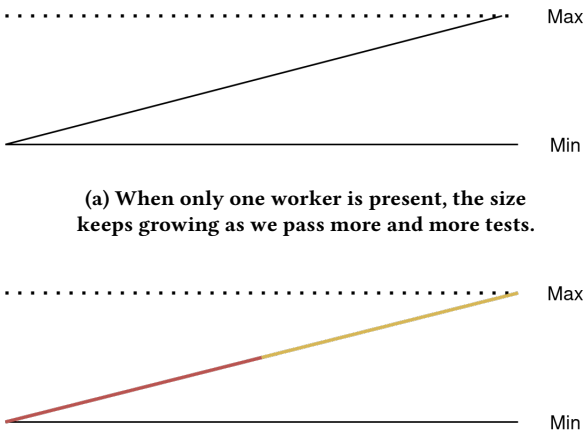
This is where the behavior of the shrink loop might return a non-deterministic result. Whereas the previous loop will always find the first counterexample in the candidate list, the parallel loop might find a counterexample other than the first one. To emulate the deterministic behavior, the new loop can choose to only signal a restart to those concurrent workers that are evaluating candidates that appeared after the current one in the candidate list, and tell them to speculatively start shrinking the new counterexample. The other workers will keep evaluating their current candidates, and if one of them turns out to be a counterexample, the current progress will be discarded, and shrinking will continue with the new counterexample. In this case, we might do some unnecessary work, but we will get the same deterministic result.

Another alternative is that when any worker has found a counterexample, all concurrent workers are restarted and told to start shrinking the new counterexample, regardless if this was the first counterexample or not. This might lead to a non-deterministic result, as the path down the rose tree of shrink candidates is not the leftmost one.

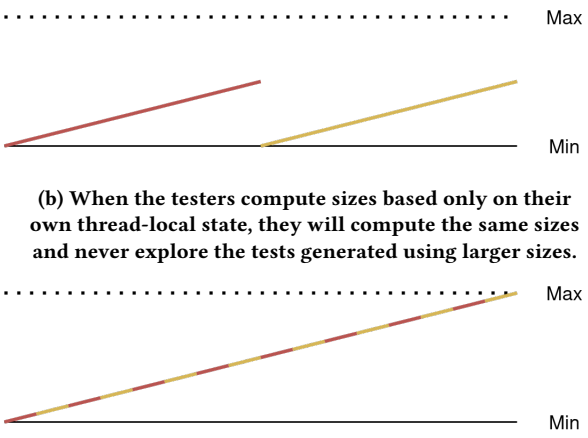
Draft note: When evaluating a property is aborted violently like this, there might be test artifacts left. There needs to exist a new combinator `graceful` that allows the developer to specify what test cleanup must be done before aborting. The authors are currently experimenting with designs for this.

5 IMPLEMENTATION

Draft note: Implementation is primarily complete, but may be adjusted slightly after some more experimentation. The full implementation will be described in the final paper. To summarize, testing is done by running multiple instances of the sequential testing loop in parallel, after modifying it to be aware of concurrent sibling testers. Parallel shrinking is implemented by augmenting the simple, existing shrinking loop such that multiple instances of it can safely and efficiently shrink the same initial counterexample.



(a) When only one worker is present, the size keeps growing as we pass more and more tests.



(b) When the testers compute sizes based only on their own thread-local state, they will compute the same sizes and never explore the tests generated using larger sizes.

(c) One solution is to give each tester a size offset. This might lead to an uneven distribution of sizes, however. In the illustration above, one tester explores all low sizes and another explores all large sizes. This uneven distribution might lead to an uneven execution time, as large tests might take longer to evaluate.

(d) Another solution is to use a stride instead of an offset. This leads to a more even distribution of tests, and hopefully a more even distribution in execution times.

Figure 1: Different illustrations of how to compute the sizes for test cases. The two horizontal bars indicate the smallest and largest size that can be computed, and the angled line illustrates what size is computed as we pass more and more tests. The size grows as we pass more and more tests.

6 EVALUATION

We evaluate QuickerCheck to answer the following five questions

- **Question one**: Is the performance of the new implementation when only one core is used comparable with the performance of the existing implementation?
- **Question two**: How much faster is the new implementation when more cores are added?
- **Question three**: Do we find bugs faster?
- **Question four**: Do we shrink counterexamples faster?
- **Question five**: What is the impact of using bound threads instead of unbound threads?

To answer these questions, we run properties and collect information. We will refer to such properties as benchmarks, and the benchmarks we use are described in the following subsection.

6.1 Benchmarks

constant. The benchmark named *constant* is supposed to indicate what the overhead of the new testing loop is. The cost of running a test is most often dominated by the time it takes to evaluate the property. This benchmark tries to eliminate the cost of evaluating the property by making this as small as possible. Thus, the property is

```
prop_constant :: () -> Bool
prop_constant () = True
```

Generating the input takes constant time, and evaluating the property takes constant time.

Compressid. This benchmark composes the two Unix commands *gzip* and *gunzip* and verifies that the composition behaves as the identity function. It generates an arbitrary string and invokes `gzip`, and passes the compressed result to `gunzip`, and asserts that the output is identical to the input. This benchmark comes in three flavors – one is a naive implementation (*naive*) that writes intermediary values directly to the file system. Since using the file system like this leads to race conditions, we also test two alternative implementations that each parallelize the property in different ways. The first, (*tmpfs*) is altered to generate fresh directories for each concurrent worker to write such files to (without changing the environment working directory), and the last one (*nofs*) uses pipes to pass values around, never accessing the file system at all.

ICT. The *ICT* benchmark tests a compiler for correct compilation. It is similar to the *Compressid* benchmark in that it is mostly doing IO, but the IO workload is heavier. Additionally, generating input for this property is more complicated. This property generates arbitrary imperative programs, compiles them, and generates code, after which the compiled binary is invoked. The result is read back and inspected. It is a concrete instantiation of the metamorphic compiler property that is listed in section 2.

Verse. This property asserts the confluence of the rewrite system for the new Verse Core Calculus [1]. A rewrite system is confluent if, regardless of which rewrite rules are applied in each state, the end result is always the same, single, normal form.

This property generates arbitrary Verse terms and applies arbitrary rewrite rules. It is a pure property that performs a moderate amount of work.

System F. This benchmark asserts the preservation property described in 2 for System F. It is similar to the Verse benchmark in that

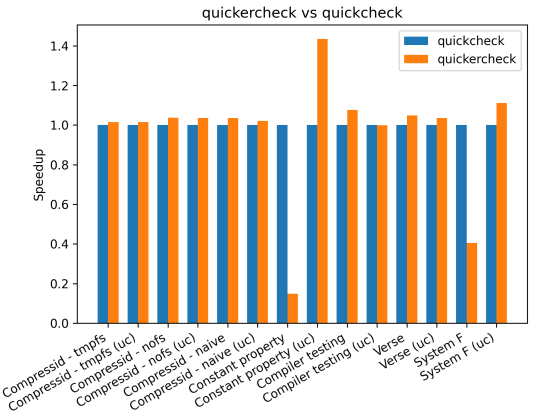


Figure 2: Illustration of the relative performance of QuickerCheck to QC when only one core is used. See figure 3 for the numbers. The two noticeable outliers in this figure are the *Constant* and *System F* benchmarks, which QuickerCheck evaluate *much* faster than QC. This is because QuickerCheck spends much less time printing the current progress.

Benchmark	Relative execution time
Compressid - tmpfs	1.02x
Compressid - tmpfs (uc)	1.02x
Compressid - nofs	1.04x
Compressid - nofs (uc)	1.04x
Compressid - naive	1.04x
Compressid - naive (uc)	1.02x
Constant property	0.15x
Constant property (uc)	1.43x
Compiler testing	1.08x
Compiler testing (uc)	1.00x
Verse	1.05x
Verse (uc)	1.04x
System F	0.41x
System F (uc)	1.11x

Figure 3: This table reports the relative execution time of QuickerCheck compared to QC when only one core is used. In this figure, a number of one indicates that there is no difference in performance between QuickerCheck and QC, and lower numbers indicate that QuickerCheck is more efficient.

it is a pure property that specifies properties of a rewrite system, but it performs less work in each test.

6.2 Results

All tests are carried out on a Dell Latitude 5400 laptop with an Intel i5-8365U (8) @ 1.600GHz CPU with 4 cores and 8 threads, 16GB of RAM, running Ubuntu 22.10. The processor turbo is turned off when running experiments.

Draft note: The authors are aware that running experiments like these on an ordinary laptop is not super interesting, and will rerun them on a machine with more resources before a final version of this paper is written.

Question one. To answer the first question we run all the benchmarks 20 times with both QC and QuickerCheck, only using 1 core for QuickerCheck. The averaged results are are presented in figure 2 and 3. All benchmarks are run both with the *chatty* flag set to `False` and `True`, as indicated by the *uc* label (*uc* meaning *unchatty*). If the *chatty* flag is set to `False`, QC will not print the current progress while testing.

Question two. To answer question two, we run all properties and measure execution time, continuously adding more cores. For each such core configuration, we run the property 20 times and compute the average time required to run all tests. These results are illustrated in figure 4, and the max speedups are reported in table 5.

Question three. To evaluate whether we find bugs faster or not, we plant bugs in the code for three of the properties. There is a certain probability that a counterexample will be generated. This probability should not change when the number of concurrent workers is increased, so the mean number of tests required to find a counterexample should remain unchanged, up to some variance. We fix the number of cores and run QuickerCheck until the bug is found while recording the execution time and the number of tests required to find the bug. We do this 1000 times per core configuration, and compute the mean number of tests required to trigger the bug, as well as the mean execution time. The mean number of tests required to find the bug with QC is rendered as a cyan point, with the mean execution time marked as an orange *x*. These results are illustrated in figures 6, 7, and 8. These measurements are collected with *chatty* set to `False`.

Question four. Draft note: The authors have not had time to properly evaluate this, but anecdotal evidence indicates that shrinking counterexamples is indeed sped up, but not necessarily as much as finding the initial counterexamples. It depends on the quality of the shrinker and how many "unsuccessful shrinks" need to be evaluated.

Question five. We run all benchmarks 20 times using unbound threads, and 20 times using bound threads. We compute the mean execution times and compare the performance of using bound threads to that of using unbound threads. The results are listed in table 9.

7 DISCUSSION

The results are generally very encouraging. As for **question one**, it appears that the cost of the new implementation is not very noticeable. The *constant* benchmark was meant to measure precisely the performance of the new loop, and it indicates that with *chatty* turned on, it is many times faster than the previous loop. This is because the new loop prints much less than the old one. If we account for this and turn off *chatty*, the new implementation is roughly 40% slower. In practice, this is most likely not a problem, as the property is already extremely fast, and thousands of tests will be executed before you can blink.

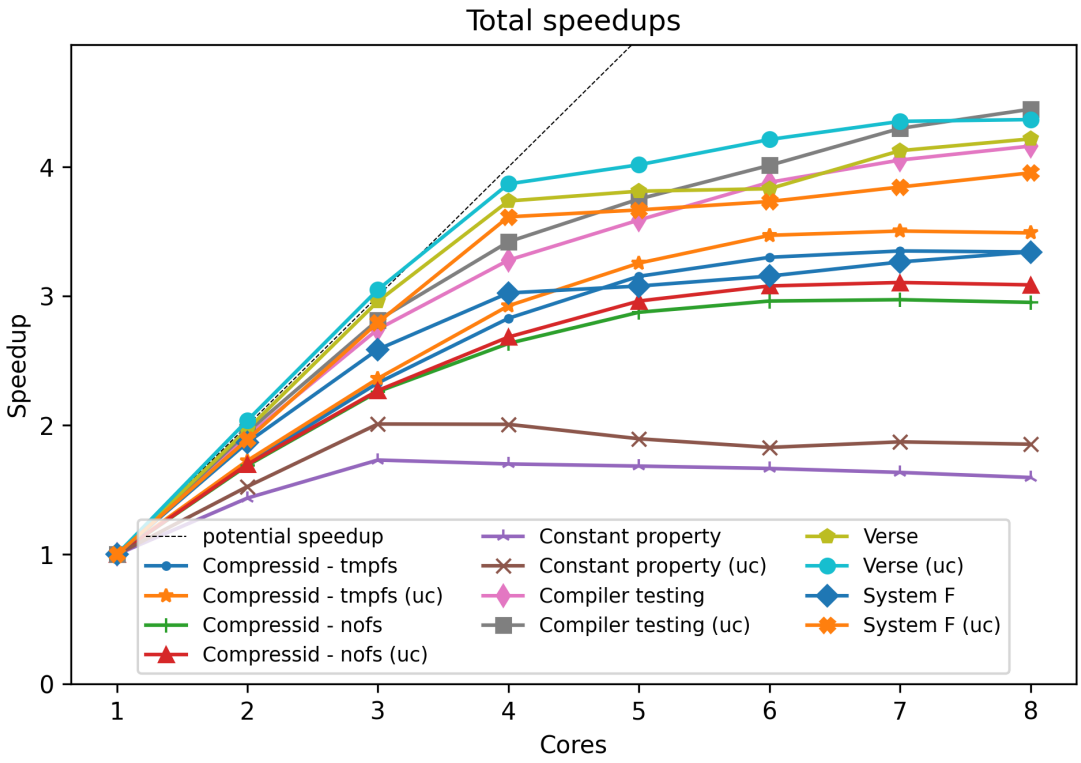


Figure 4: Illustration of the speedups acquired when more cores are added, for every benchmark.

Benchmark	Max speedup
Compressid - tmpfs	3.35x
Compressid - tmpfs (uc)	3.50x
Compressid - nofs	2.97x
Compressid - nofs (uc)	3.11x
Constant property	1.73x
Constant property (uc)	2.01x
Compiler testing	4.16x
Compiler testing (uc)	4.45x
Verse	4.22x
Verse (uc)	4.37x
System F	3.34x
System F (uc)	3.95x

Figure 5: Maximum speedups acquired for the different benchmarks as more cores are added.

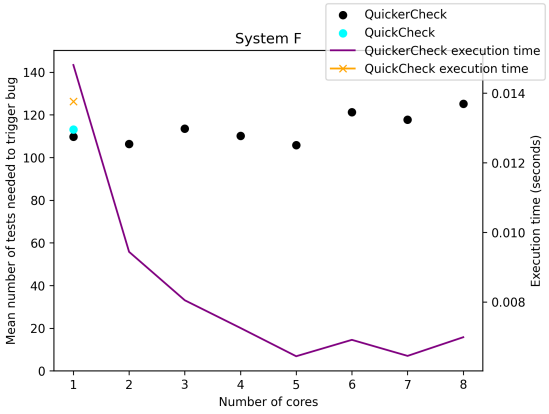


Figure 6: The property that asserts preservation for system F is sped up by a factor of slightly more than two. It is worth noting here that the execution time is very low, so the speedup might be hindered a bit by the startup cost of the parallel testing loop.

Looking at more realistic properties doing some actual work, the penalty of the new implementation is consistently below 10%, oftentimes lower.

Moving beyond the case where N = 1, QuickerCheck quickly outperforms QC when more cores are added. The performance scales very well until the point where hyperthreading takes over, where acceleration drops. The answer to **question three** seems to be *yes*.

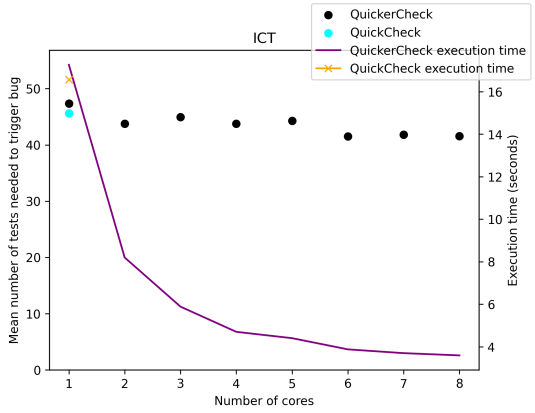


Figure 7: The time it takes for the metamorphic compiler testing property to discover the bug is sped up by a factor of more than four. From more than 16 seconds to less than four.

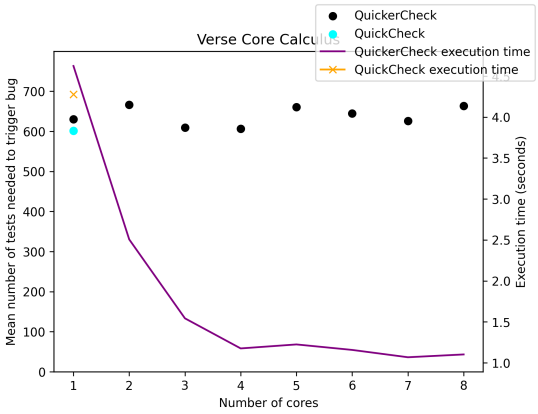


Figure 8: The property that asserts confluence for the Verse Calculus is sped up by a factor of roughly four.

The results indicate that bugs are indeed found faster, answering **question two**. The mean number of tests required to find a counterexample remains the same (up to some variance), while the time it takes to run those tests drops. This is very encouraging for cases where QC has a limited amount of time to run, e.g. as part of an automated test script.

There seems to be a performance penalty to using bound threads instead of the default unbound ones. Most benchmarks did not experience any noticeable difference (+-1%), but two of the benchmarks experienced a 5% and 9% slowdown on average. This supports the choice of using unbound threads by default, but we do still not know why precisely these two properties experienced a slowdown. Further evaluation of this would be interesting.

Draft note: The discussion will be extended to include shrinking, related work and argue for why what QuickerCheck does is better than just launching multiple instances of QC and running entire properties in parallel.

8 CONCLUSIONS

REFERENCES

[1] Lennart Augustsson, Koen Claessen, Simon Peyton Jones, Guy L. Steele Jr., Joachim Breitner, Ranjit Jhala, Olin Shivers, and Tim Sweeney. 2023. The Verse Calculus: a Core Calculus for Functional Logic Programming. *Proc. ACM Program. Lang.* ICFP (2023).

[2] Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000*, Martin Odersky and Philip Wadler (Eds.). ACM, New York, NY, USA, 268–279. https://doi.org/10.1145/351240.351266

[3] John M. Hughes and Hans Bolinder. 2011. Testing a Database for Race Conditions with QuickCheck. In *Proceedings of the 10th ACM SIGPLAN Workshop on Erlang* (Tokyo, Japan) *(Erlang '11)*. Association for Computing Machinery, New York, NY, USA, 72–77. https://doi.org/10.1145/2034654.2034667

[4] Simon L. Peyton Jones, Andrew D. Gordon, and Sigbjørn Finne. 1996. Concurrent Haskell. In *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996*, Hans-Juergen Boehm and Guy L. Steele Jr. (Eds.). ACM Press, 295–308. https://doi.org/10.1145/237721.237794

[5] Barton P. Miller, Lars Fredriksen, and Bryan So. 1990. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM* 33, 12 (1990), 32–44. https://doi.org/10.1145/96267.96279

Benchmark	1 cores	2 cores	3 cores	4 cores	5 cores	6 cores	7 cores	8 cores	mean
Compressid - tmpfs	0.99x	0.99x	1.01x	0.98x	0.99x	1.00x	1.01x	1.00x	1.00x
Compressid - tmpfs (uc)	1.01x	1.03x	1.01x	1.00x	1.01x	1.00x	1.00x	1.00x	1.01x
Compressid - nofs	1.00x	1.02x	1.00x	0.99x	1.00x	1.01x	1.01x	1.01x	1.00x
Compressid - nofs (uc)	1.00x	1.02x	1.00x	0.99x	1.00x	1.01x	1.02x	1.03x	1.01x
Constant property	1.00x	1.01x	1.03x	1.13x	1.18x	1.06x	1.00x	1.02x	1.05x
Constant property (uc)	1.00x	1.00x	1.02x	1.03x	1.45x	1.14x	1.09x	0.99x	1.09x
Compiler testing	1.00x	0.99x	1.00x	1.00x	1.00x	1.00x	1.01x	0.99x	1.00x
Compiler testing (uc)	1.00x	1.00x	1.00x	1.00x	1.01x	1.00x	0.98x	1.00x	1.00x
Verse	0.99x	1.03x	1.01x	1.02x	1.00x	0.99x	1.01x	0.99x	1.01x
Verse (uc)	1.01x	0.99x	0.98x	1.00x	0.99x	1.00x	1.03x	1.01x	1.00x
System F	0.99x	1.01x	1.00x	1.00x	0.98x	0.98x	0.99x	1.00x	1.00x
System F (uc)	1.00x	1.00x	1.01x	0.99x	1.00x	1.00x	1.00x	1.00x	1.00x

Figure 9: The relative performance of using unbound or bound threads. Numbers which are less than one indicate that using a bound thread was faster, while a number greater than one indicate that using a bound thread was slower. The *constant* property suffered the most from using bound threads, with the mean penalty being a 9% slowdown. However, the reported penalty fluctuates a lot as cores are added. For instance, when using 5 cores, the penalty is a 45% slowdown!