Cloud-Based Verification of Concurrent Software

Gerard J. Holzmann*

NASA Jet Propulsion Laboratory,
California Institute of Technology, Pasadena, CA, USA
gh@jpl.nasa.gov

Abstract. Logic model checkers are unparalleled in their ability to reveal subtle bugs in multi-threaded software systems. The underlying verification procedure is based on a systematic search of potentially faulty system behaviors, which can be computationally expensive for larger problem sizes. In this paper we consider if it is possible to significantly reduce the runtime requirements of a verification with cloud computing techniques. We explore the use of large numbers of CPU-cores, that each perform small, fast, independent, and randomly different searches to achieve the same problem coverage as a much slower stand-alone run on a single CPU. We present empirical results to demonstrate what is achievable.

Keywords: Software verification · Logic model checking · Software Testing · Concurrency · Multi-threaded code · Cloud computing · Swarm verification · Massive Parallelism

1 Introduction

Although the amount of memory that is available on standard desktop computers continues to increase, clockspeeds have stalled at their current levels for well over a decade. With access to ever larger amounts of RAM memory, logic model checkers could in principle search ever larger problem sizes, but the time required to perform those searches can become substantial. Time, not memory, has become the main bottleneck in formal verification.

As one simple example, consider the CPU-time requirements for the verification of a large problem on a system with 128 GB of main memory. Even if the model checker runs at a fast rate of about 10^5 newly discovered states per second, we can explore only about 9.10^9 states in a 24-hour period.

Using the bitstate storage method [1, 6, 8], we can record up to 10^{12} unique hash signatures of states in 128 GB, cf. Appendix A. This means that a verification run could take over three months before it fills RAM.

At this point, the search may still be incomplete. Assume it has reached a coverage of $C\%$ of the full search space. There are two things we may want to do:

* This research was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

increase the coverage until it reaches 100%, or obtain the same level of coverage faster by performing the search differently, using more machines. We show that both objectives can be realized with the help of a cloud-compute infrastructure and hash-randomization techniques.

To see how this could work, first note that when, in our example, we reduce the size of the hash-arena from 128 GB to 128 MB, we can store 10^3 times fewer states, and as a result the search would also complete up to 10^3 times faster. While it is attractive to see the maximal runtime shrink, we do not like the simultaneous reduction in coverage. But we can fix that.

If we repeat the faster run with a randomly different, statistically independent, hash-function, the new run will take roughly the same amount of time as before, but visit a partly *different* set of states. The two runs together will have greater coverage then either one separately, but they take the same amount of time and since they are independent they can be performed in parallel.

As first noted in [6], we can continue to increase the cumulative coverage of the verification, until it reaches 100%, by performing more and more independent runs, each time using a different, randomly selected hash-function.

Since each faster run, using 128 MB instead of 128 GB, explores 10^3 fewer states, we may expect that we would need to perform well over 10^3 independent fast runs to make up for the loss of coverage, because inevitably there will be overlap between the state spaces that are explored in the separate runs. The question is how many more parallel runs would we need to bridge the gap. We study that question in this paper.

Earlier work, e.g., [4, 9], considered the use of only small numbers of CPUs in multi-core systems or local networks. Here we focus on the potential increase in capability that can result when we make use the massive parallelism that is available today in commercial cloud compute networks.

2 Methodology

We make use of the bitstate storage method [1, 6, 8] to perform multiple independent partial searches in large state spaces. The method we describe here depends on a capability to generate randomly different hash polynomials to perform the individual searches. The Spin model checker [16], Version 6.4, was extended with an algorithm for generating such polynomials. The algorithm uses an unsigned 32-bit number as a seed for a scan for a usable hash polynomial that completes in a fraction of a second. The use of a 32-bit seed trivially sets an upper-limit to the number of distinct hash polynomials that could be generated to 2^{32}. Not every seed number leads to a usable polynomial in the first round of the scan though, so the true limit is lower.

Figure 1 shows the results of a test of the number of tries that the hash-polynomial generation algorithm makes, starting with random 32-bit numbers. In this tests, the number of tries was between zero and 200, with an average of about 30 tries. This means that an upper limit for the total number of distinct

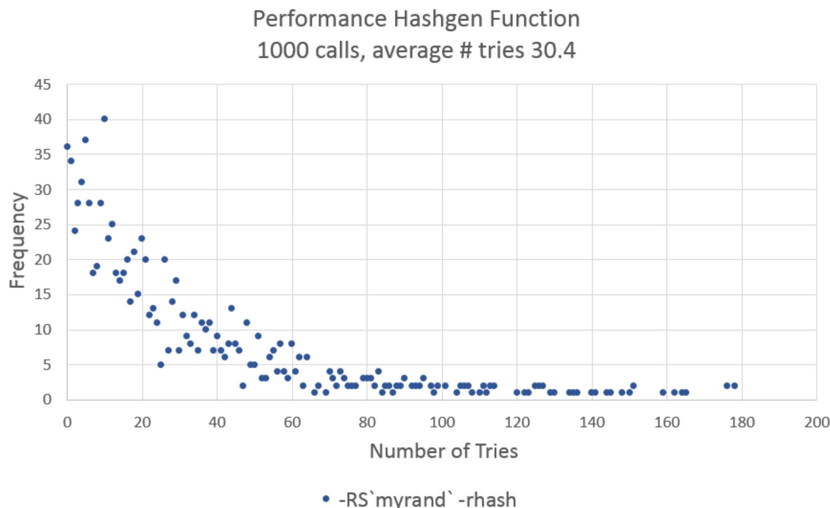

Fig. 1. The number of tries that the hash polynomial generator makes, starting from a randomly chosen unsigned 32-bit number. The average over 1,000 calls was 30.4 tries.

hash polynomials that the algorithm that we use could generate is in the order of 10^8.

We can increase the diversity of parallel runs further by varying additional search parameters. We can do so, for instance, by randomly changing also the number of hash-polynomials that are used to compute the hash-signature of each state, changing the maximum search depth, changing the search order (considering processes and transitions from left to right in the transition table, right to left, or in random order), and the search discipline itself (e.g., depth-first, breadth-first, or a context-switching bounded search [10]). Randomization and diversification can thus be used to create very large numbers of independent verification engines.

In the measurements we report on in this paper we restrict to just a few of these possible search options. To perform the tests, we used the following runtime flags and parameters that the current version of Spin (Version 6.4.4) provides.

`-RS`N to seed the random number generator to N
`-rhash` to randomly generate a bitstate hash polynomial
`-w`N to set the size of the hash-arena to 2^N bits
`-k`N to set the number of bits set per state to N
`-m`N to set the maximum search depth to N

To seed the random number generator we avoid using timestamps or the the standard `rand` command from Unix and Linux-based systems. The Unix `rand`

command uses a time-based algorithm, which means that it can return the same value if called too quickly in succession. We can avoid this by defining the following command, as a more reliable source for random numbers on Linux and Linux-like systems:

```
$ cat /usr/local/bin/myrand
#!/bin/bash
od -vAn -N4 -tu4 < /dev/urandom | sed "s; *;;"
```

We use this command to start potentially large sets of verification runs that each are guaranteed to receive a different seed value, for instance in a Bourne or Bash shell-script as follows:

```
$ spin -a spec.pml
$ cc -O2 -DBITSTATE -DRHASH -o pan pan.c
$ for i in ‘seq 1000‘
  do  # start each run in the background
   ./pan -RS‘myrand‘ -rhash -w20 -k1 -m20000 &
  done
```

The compiler directive `-DBITSTATE` is used here to enable the bitstate storage method, and directive `-DRHASH` enables runtime option `-rhash`, which not only triggers the generation of a random hash polynomial but also randomizes the transition selection order to be used as well as the process scheduling orders, to achieve greater diversity.

The argument `-w20` sets the hash-arena size to 2^{20} bits (128 Kbytes) for a very fast model checking run. Argument `-k1` sets the number of bits set per state to one, and `-m20000` sets the maximum search depth to 20,000. The numeric argument to each of these last three arguments can, of course, readily be randomized as well.

3 Measurements

We explore two types of applications of the cloud-based verification approach we discussed above. First we consider the typical case where the purpose of a model checking run is to demonstrate the *presence* of subtle defects in a multi-threaded software application that may be missed in standard testing. A cloud-based verification method could provide an added capability that is both faster and more thorough. Separately, we also consider the potential use of this technique for formally proving the *absence* of defects.

3.1 Defect Discovery

The initial motivation for this study was the verification of the source code for a double-sided queue algorithm using compare-and-swap instructions, referred to

as the DCAS algorithm. The implementation described in [2] has a subtle bug, which was discovered when a formal proof of correctness was attempted with the PVS theorem prover [3]. The discovery of the bug took several months of work with the theorem prover.

We are interested here in finding out if the same bug can be discovered faster with a cloud-based verification approach.

We discussed the verification of the DCAS algorithm earlier in [11]. We used Modex [14] to extract a Spin model from the C code, while adding two test drivers, also in C, that would normally be part of a standard test suite. In [11] our objective was to show that we can extract formal models from unmodified C code, while adding only the test code that a tester would normally already provide. The model checking runs can then be automated to significantly increase the accuracy of the tests that are performed.

The verification described in [11] immediately discovered an assertion violation, with just two threads of execution in the DCAS model: a reader and a writer. This was unexpected because the only defect in the algorithm known at that point required at least three threads of execution to complete. Unfortunately, the test drivers we used were faulty, and the assertion violation that was found originated in the test itself, not in the code being tested. A corrected version of the test drivers is given in [12].

With the corrected version of the test drivers, the model checker needs just three seconds of CPU-time to formally prove that with two threads of execution the DCAS algorithm is correct and cannot violate the assertion.

Increasing the number of threads from two to three increases the complexity of verification sufficiently that an exhaustive proof becomes intractable. We aborted a bitstate verification attempt after 138 hours of CPU-time, having explored over 10^{11} states, without locating the bug. The question is if a cloud-based swarm of small and randomly different verification tasks can still succeed in locating the bug, and do so quickly and reliably.

As noted, by setting the size of the hash-arena we can control how long the verification tasks can maximally take. Figure 2 illustrates this effect, for the DCAS verification. To check if a fast cloud verification run can identify the flaw in the algorithm we set the size of the bit-state hash-arena to just 128 KB, using runtime parameter `-w20`. We performed a verification with a swarm of 1,000 small independent verification tasks, differing only in the selection of the hash-function each used. The script below describes the verification steps.

Compared to the earlier examples, we added the compiler directive `-DPUTPID` to add the process number of each thread of execution to the name of any counter-example traces that are found, to prevent the parallel runs from overwriting each other's output. We also added the directive `-DSAFETY`, since we're only interested in assertion violations for this example. Note that the verifier is linked to the C source code of the implementation.

```
$ spin -a dcas.pml
$ cc -O2 -DBITSTATE -DSAFETY \
        -DRHASH -DPUTPID -o pan pan.c dcas.c
```

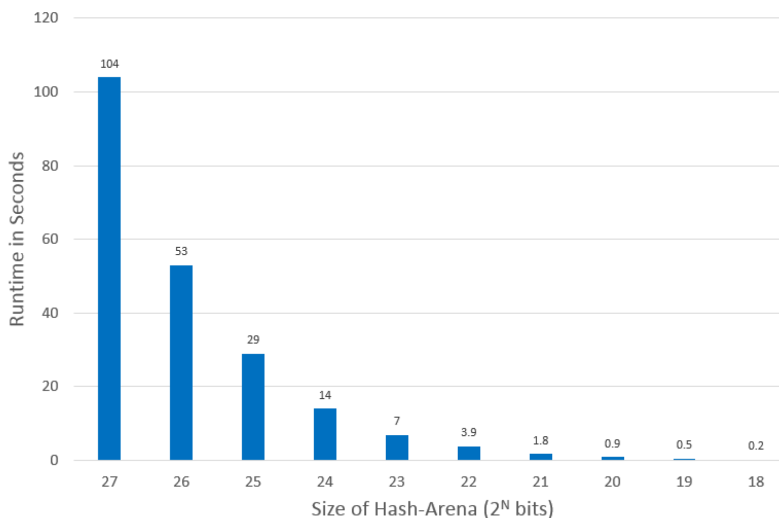

Fig. 2. Correlation between the size of the hash-arena and runtime. A choice of N=20 corresponds to a hash-arena size of 2^{20} bits or 128 KB.

```
$ for i in `seq 1000`
do   ./pan -w20 -RS`myrand` -rhash -k1 &
done
```

Each of the verification runs uses a hash-arena of just 128 KB of memory, and sets one single bit per state visited. Because of the very small hash-arena used, each run completes in a fraction of a second, which means that if all runs are performed in parallel the entire task also completes that quickly.

In this test, two of the 1,000 runs successfully located the assertion violation in the code and generated counter-example traces (Table 1), in a fraction of a second.

If we increase the size of the hash-arena, more runs will be able to locate a counter-example, but the runs will also take longer to complete. If, on the other hand, we decrease the size of the hash-arena, a smaller fraction of the runs can be expected to successfully identify the bug in the algorithm. We can counter that effect by increasing the size of the swarm. To illustrate this, we performed sequential runs of the verifier at a range of different settings of the hash-arena size, stopping each batch as soon as the first counter-example was found (with that hash-arena size), see Figure 3.

As expected, the number of runs to be performed increases as the size of the hash-arena shrinks. Clearly there is a minimum size of a swarm of parallel tasks before the swarm as a whole can be successful in locating defects. Trivially, if each task has a probability p of locating a defect, we would need to run at least $1/p$ parallel tasks to be successful.

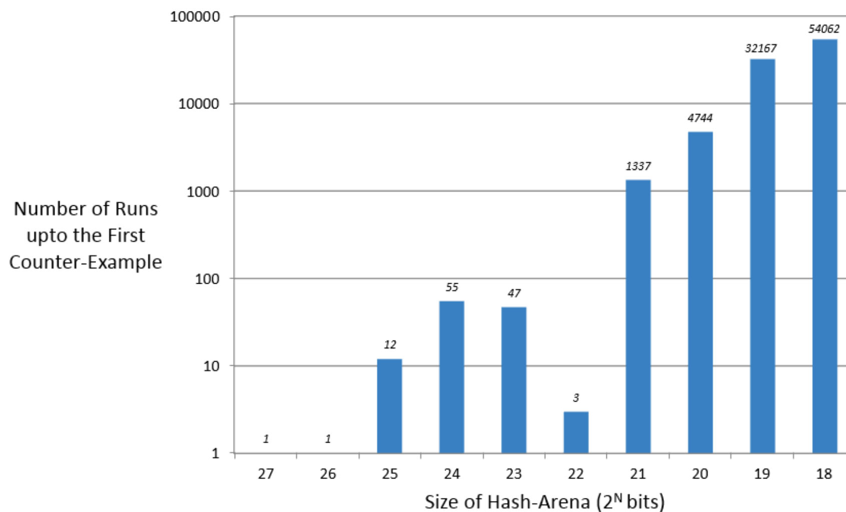

Fig. 3. Correlation between the size of the hash-arena and the number of random runs needed to find a first counter-example. From 16 MB (2^27 bits) to 32 KB (2^18 bits).

Figure 3 shows the randomness in this process. In these tests it took closer to 5000 runs at a hash-arena size of `-w20` to uncover the first counter-example, where in our earlier test fewer than 1,000 runs sufficed. Similarly, for a hash-arena size of `-w22` or 512 KB, this test reported a first counter-example after just three attempts, where based on context perhaps a run of 300 would statistically have been more likely. Note that all runs with a hash-arena size smaller than `-w26` (16 MB) completed in under a minute of runtime and runs below `-w20` in under a second (cf. Figure 2). In both cases this is considerably faster than the weeks or months that would be needed for a traditional exhaustive verification attempt.

Other Tests. We performed two other tests of this technique on similarly large models that defy exhaustive verification by traditional means, summarized in Table 1.

The first of these is the Fleet model, which is a pure Promela model of a distributed system architecture designed by Ivan Sutherland [17]. A cloud-based verification attempt using just 100 small parallel bitstate runs, similar to the runs used for the DCAS model, locates multiple variants of the bug (also an assertion violation), again in a fraction of a second. In this case, setting the hash-arena as low as `-w13`, storing just 2^{13} bits (or 1 Kbyte of memory), sufficed to locate the bug.

In another test we applied the cloud-based verification method to test the C implementation of POSIX routines implementing a non-volatile flash file system (NVFS) designed for use in a spacecraft, cf. [13]. The models were extracted

Table 1. Performance of Cloud Verification Runs. In each of the cases considered, the verification tasks are too complex for traditional exhaustive verification to succeed in a reasonable amount of CPU-time, but the cloud verification approach succeeds in locating counter-examples quickly.

Model	Nr. Cores	Run-Time	Defects found
DCAS	1000	0.2s	2
Fleet	100	0.1s	1
NVFS	500	0.5s	6

from the C code in this case with the Modex model extractor, [7, 14, 11]. The resulting models are well beyond the reach of traditional verification techniques. In bug-finding mode we ran 500 parallel tasks with a hash-arena sized to store 2^{17} bits (16 Kbytes) of memory, which sufficed to locate six counter-examples of the correctness properties in under one second of runtime.

3.2 Formal Verification

The cloud-based verification approach can be a powerful alternative for bug hunting, since this generally does not require exhaustive coverage of a problem domain. This does not necessarily mean though that it is also be suitable for full formal verification. One important reason for this is that there is no easy way to determine what the cumulative coverage is that is realized by a swarm of small fast verification tasks. On statistical grounds we know that larger swarms provide better coverage than smaller ones, and that swarms using larger hash-arenas similarly provide better coverage than those using smaller hash-arenas.

Measuring the cumulative coverage can be done by instrumenting the code, capturing every state visited by every task, and then post-processing that information to compute the total number of unique states that was explored by all tasks together (i.e., after eliminating the overlap). If we know how many reachable system states there are, we can then compute the cumulative coverage accurately.

If we do not know the total, we can in deduce an approximate coverage by performing two or more swarm runs sequentially, with different hash-arena sizes for each swarm run. Figure 4 illustrates the effect on the number of unique states that is visited when we perform this experiment for a range of different hash-arena sizes.

Note that once 100% coverage is realized, the number of unique states visited can no longer increase when we increase the size of the hash-arena: the growth ratio will be one. When we are far from complete coverage, on the other hand, the number of unique states visited will double with each doubling of the size of the hash-arena: the growth ratio is now two. If we are somewhere in between these two points, the growth ratio will be somewhere between one and two. The growth ratio measured can thus give us an indication of the problem coverage realized, though not a precise one.

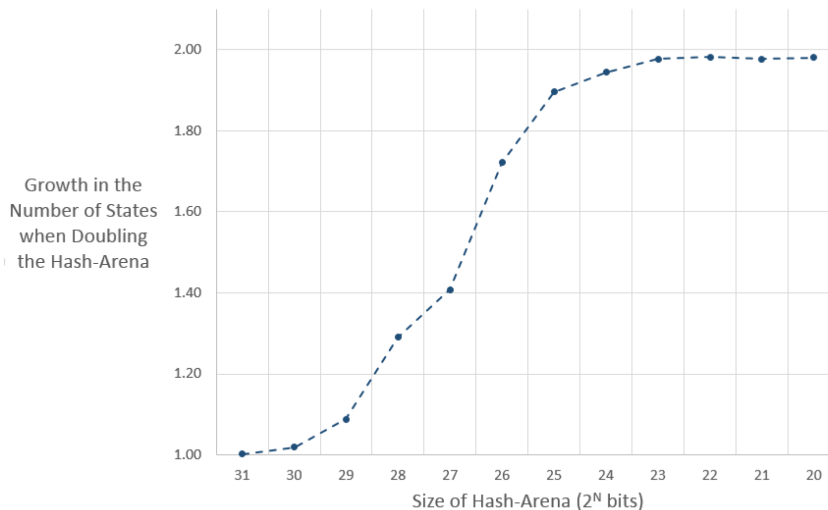

Fig. 4. Growth in the number of unique states visited per run, when the hash-arena size is doubled, for a range of hash-arena sizes. Model from Example 1 in Section 3.2, also used in Fig. 6, [5].

Adding the instrumentation and the processing of the counts will of course lead to the loss of a portion of the speed advantage of a swarm verification search.

For the following tests we performed we chose test examples for which we could measure the cumulative state space size independently with standard verification techniques, so that we could calculate the cumulative coverage of a cloud-based verification run accurately.

We use two medium-sized examples with known state space sizes. The first is an algorithm for concurrent garbage collection [5], with a total of about 192 Million reachable system states. The second is a Spin model of an operating system for small spacecraft [15], with about 22 Million reachable system states.

We setup a test environment for the measurements on a machine with 64 cores and 128 GB of shared memory. We run a single *collector* process on this machine that creates a traditional hash-table, large enough to store all reachable states. The *collector* process creates an array of n buffers (channels) in shared memory, as many as fit, to receive 64-bit state hash signatures from n different *generator* processes that run in parallel.

To test the coverage that is realized by a swarm of N small verification tasks, where generally N is much larger than n, we split up all tasks in n batches of N/n tasks each. The tasks in each batch execute sequentially, and are instrumented to connect to a specific channel to the collector when they start up, to transmit the hash signatures for the states that each of those tasks explore. The *collector* reads the hash signatures from the channels, and adds them to its own state hash-table, while omitting duplicates. At the end of the experiment, the collector can

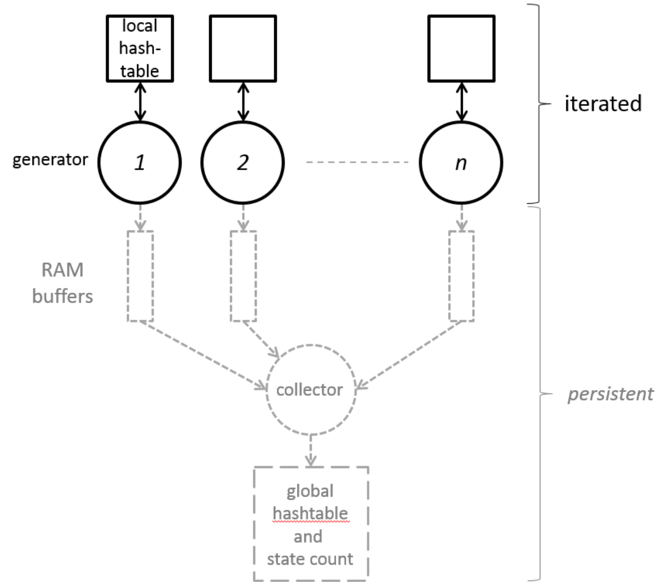

Fig. 5. Test Setup for Measuring Cumulative Coverage of Cloud-Verification

give us a count of the number of unique states that were reported by all tasks cumulatively, so that we can compute the coverage of all N *generator* processes combined. The test setup is illustrated in Figure 5.

Since each task in the swarm is configured to use a randomly generated hash-polynomial, we have to make sure that the hash signatures that are sent to the *collector* process are computed with a single additional polynomial that is fixed for all verification tasks. To make the measurement possible, the *generator* processes therefore must compute an additional fixed hash-signatures for each state, which slows them down compared to an actual cloud-based verification run.

Example 1. The Spin model we used for this first example was adapted from a formal PVS specification written by Klaus Havelund [5]. As was the case with the DCAS algorithm we discussed earlier, the PVS proof of this concurrent garbage collection algorithm took months of dedicated human time to complete.

In this first example, the search space that must be explored for an exhaustive verification attempt is very deep. A depth-first search attempt, for instance, quickly reaches a depth of 50 Million steps. This feature makes the application of a cloud verification especially interesting, because it may limit the accuracy of each small verification task.

To get an accurate measurement of the full state space size for this model we performed an exhaustive verification with Spin. With compression enabled, the verification requires about 23.2 GByte of memory, and reaches about 192

Million states in 17 minutes of CPU-time. Note that a cloud based approach is not needed for this model, but we chose it to study the relative performance of this approach.

We tested the coverage for hash-arena sizes that range from a low value of 1 MByte per task (using runtime parameter `-w23`), or about four orders of magnitude smaller than the 23.2 GByte that is needed for a single exhaustive verification, to 256 MB (`-w31`), or about two orders of magnitude below the size needed for exhaustive verification. We measured the cumulative coverage for swarms of verification tasks ranging from one single task to 1,048,576 small parallel tasks. Figure 6 summarizes the results.

At the lower end of the scale, a run with parameter `-w23` completes in about 20 seconds of CPU-time, or about 50 times faster than the exhaustive run. A run with parameter `-w25` takes four times as long: about 80 seconds, about 12 times faster than exhaustive.

As expected, the further we shrink the size of the hash-arena, the more cores are needed to make up for the loss of coverage. The increase is exponential. A first set of say 1,000 cores provides most of the problem coverage. Increasing coverage closer and closer to 100% becomes increasingly expensive for shrinking hash-arena sizes (which is needed to produce faster verification results). For `-w28`, Table 2 shows how quickly the number of cores needed to reach specific levels of coverage increases.

Table 2. Number of cores needed to reach specific levels of coverage for the verification of the concurrent garbage collection algorithm, using a hash-arena of 32 MB (`-w28`), which is 0.1% of the size needed for a single exhaustive verification run.

Nr Cores	Coverage
1	69.7%
3	90%
8	99%
32	99.9%
64	99.99%
256	99.999%
512	99.9999%
2048	100%

We can speed up the verification eightfold by shrinking the hash-arena further to `-w25`, but we will then need 2,048 cores to reach 90% coverage. To increase coverage to 98.7% requires 65,536 cores. Substantially more cores would be needed to reach 100% coverage for this very small hash-arena, but it should still be feasible.

Because all verification tasks are independent, they can be run either in parallel or sequentially, on any available machine. If we have access to a large cloud network, it is advantageous to execute all tasks in parallel for the fastest

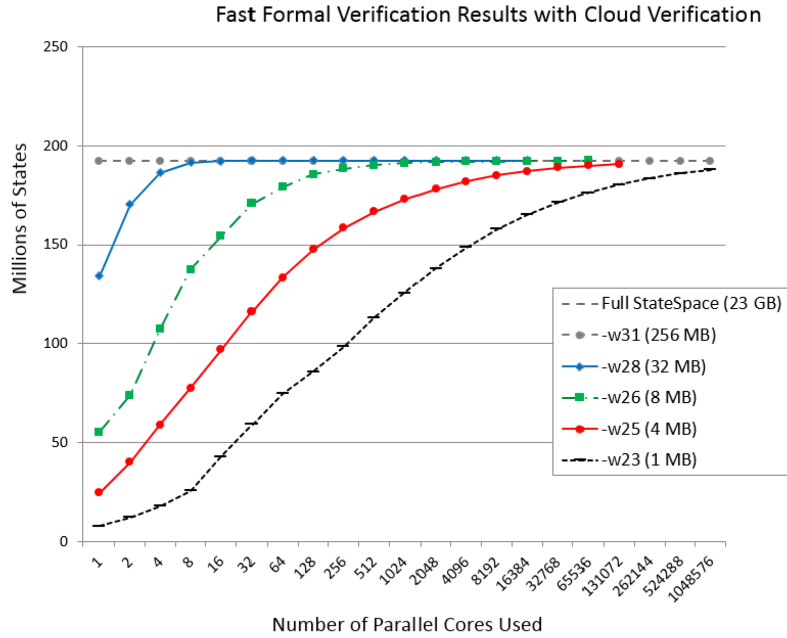

Fig. 6. Number of unique states reached, out of 192 Million known reachable states, for varying hash-arena sizes and varying numbers of cores, for a Spin verification model of a concurrent garbage collection algorithm [5]. Storing a compressed version of the full statespace takes 17 minutes and 23 GB. The same coverage can be realized using a Cloud network in 20 seconds using four orders of magnitude less storage (1 MB).

possible result. If, however, we only have access to a single machine with limited memory that is not large enough to support a single full exhaustive verification run. We can improve the problem coverage by performing a large number of randomized bitstate runs sequentially, although the speed advantage would be lost in that case.

Example 2. As a second example we look at a Spin model of the DEOS operating system kernel developed at Honeywell Laboratories, discussed in [15]. The results of a series of experiments are shown in Figure 7.

Reading the chart in Figure 7 vertically: when given the same amount of *memory*, the cumulative coverage reached by 100 randomized parallel runs is clearly superior to that of a single run.

Reading the chart horizontally, we can see that the same *coverage* can be realized by a parallel swarm using up to 16 times less memory per task (2^4), and thus also 16 times faster.

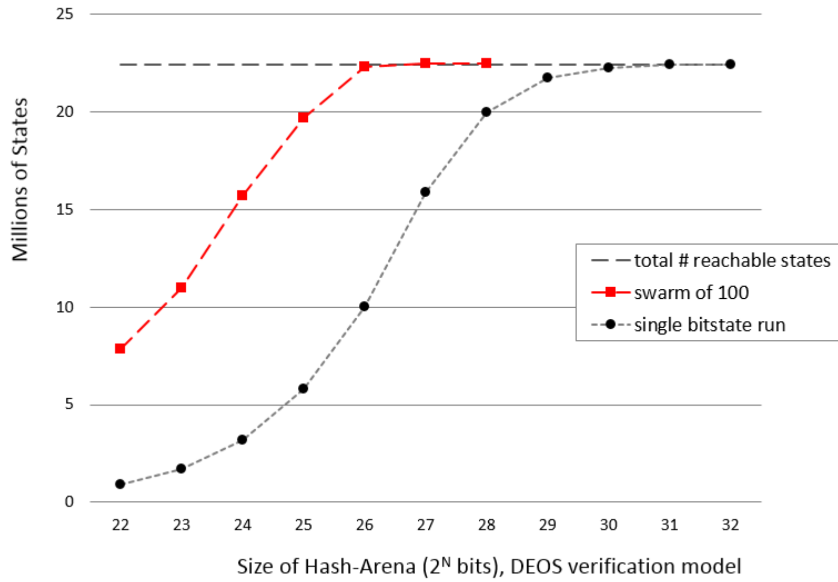

Fig. 7. State Coverage of a swarm search for the DEOS verification model. In this experiment we used 100 randomized parallel bitstate runs, and compared the coverage realized with that of a single bitstate run, as a function of the amount of memory used. When sufficient memory is available (right-side of the chart) both methods perform similarly. When less memory is available, though, the swarm approach achieves greater coverage. Every increment (horizontal axis) indicates a doubling of the hash-arena size.

4 Comparisons

To understand better what the relative performance of a cloud-based verification approach is relative to standard software testing, we measured the number of unique system states that is reached in a rigorous software test of one module of the flight code for a recent space mission: the NVFS file system code that we saw earlier in Table 1.

The standard test that was used for this application was designed to realize 98% MC/DC coverage, which is in line with the prevailing guidance documents DO-178B and DO-178C from the RTCA (Radio Technical Commission for Aeronautics) for safety and mission critical software systems. We independently measured that a total of 35,796 distinct system states were reached in these tests, exploring approximately 100 unique execution paths. These tests and the corresponding execution paths were of course not randomly chosen, but designed to test nominal and moderately off-nominal executions of the code, to make sure they comply with all requirements, cf. Figure 8.

For comparison, we also performed a full model checking run of the same code, and measured that it reached a total of 745 Million unique system states,

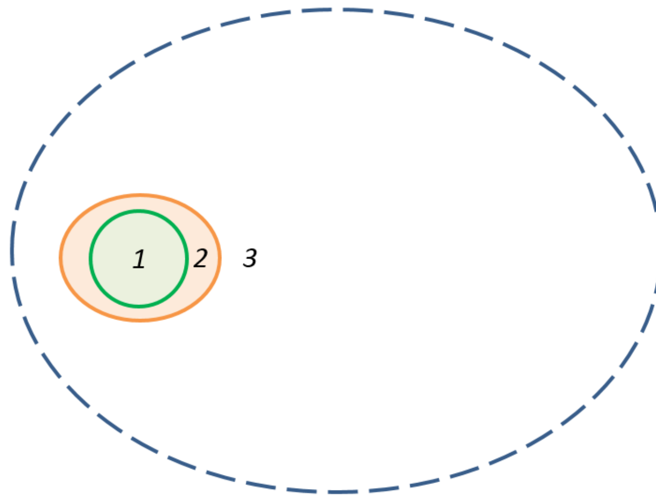

Fig. 8. Set 1 illustrates the set of nominal executions that is normally tested most thoroughly. Set 2 is a small subset of off-nominal executions that are often included in a system test. Set 3 contains all other executions, which is generally dominated by the off-nominal behaviors. In the example of the NVFS file system tests, sets 1 and 2 combined were shown to cover just 0.01% of all possible behaviors. A randomly chosen execution is most likely to fall in set 3, and is therefore likely to reveal untested erroneous behaviors.

while exploring about 50 Million unique execution paths. This means that the standard test provided a coverage four orders of magnitude smaller than the model checking run, or just 0.01%. In this context it is not surprising that the model checking run was able to uncover a subtle flaw in the code that was missed in the standard software test suite.

This example illustrates what the added value can be in a cloud-based verification approach, even when it falls short of realizing 100% problem coverage. Consider, for instance, the case where a cloud-based verification achieves only a low level of coverage of just 10%. Even this low coverage level is still three orders of magnitude better than a typical software test suite.

The two tests are not immediately comparable by their relative levels of problem coverage though. The difference is that the 0.01% executions form the standard test are carefully chosen to include most nominal and moderately off-nominal executions, while the cloud-based verification samples 10% of the execution space randomly, independent of how nominal or off-nominal the selected executions are.

The two sets, therefore, do not necessarily overlap, so it would not be valid to say that a cloud-based approach could replace standard software testing.

It is equally invalid, though, to say that unless we can reach exhaustive problem coverage no intermediate step is of value. It should be clear that even

the coverage delivered by a cloud-based verification approach near the left-hand side of Figure 6, with limited problem coverage, can *significantly* increase the chances of finding subtle software bugs that would otherwise escape detection, or failing to do so increase our confidence that such bugs do not exist.

5 Conclusions

We have explored the characteristics of a new method for overcoming the computational complexity of software verification problems, by using *massive parallelism*. This approach is motivated by the expectation that that access to large cloud compute networks is quickly becoming routine. Our measurements show that this approach can increase the rigor of software testing in a meaningful way.

Several heuristic strategies are possible to increase the chances of finding software defects in the shortest possible amount of time, given the availability of a fixed maximum number of CPU cores. The simplest strategy would be to pick the desired maximum runtime, and size the hash-array for each verification task the same, so that all complete in roughly the same amount of time. We can, however, also decide to do this only with half of all available CPU cores. On half of the remaining cores we can then give each task only half the size of the hash-array compared to the first set, which makes them complete twice as fast, while providing half the (randomly selected) coverage. We continue that process, each time taking half the number of remaining available cores, and halving the hash-array size. This method trades some coverage for an increased chance to find defects in the shortest amount of time. It is similar to the iterative search refinement strategy that was first described in [7], but with all phases of the search running in parallel, rather than one by one.

With a response time of minutes or less, there is little impediment to perform series of experiments on a cloud network that will either result in defect discovery, or the best available confidence for the absence of defects. In the cases considered here, an investment in this type of cloud-based verification compares favorably with all currently existing alternative techniques, both those that are more rigorous and those that are less so.

A Appendix: Bitstate Hash Storage

The bitstate hash storage method [6] can most easily be understood as an application of the theory of Bloom filters [1]. The storage method is used in explicit state on-the-fly model checkers, such as Spin [8], for fast approximate searches in statespaces that are too large to be explored exhaustively. The method uses one or more hash functions with uniform random distribution to compute 64-bit signatures of each newly generated reachable state.

A hash-arena of, say, 32 GB (2^{35} bytes) contains 2^{38} addressable bits. If using a single hash-function, we can use the 64 available bits from the hash signature to produce a 35-bit address into the hash-arena. If the bit at this address is at its initial value of *zero*, then we know that the newly generated state was not visited

before in the search. We now record the fact that the state has been explored by setting that bit position to a *one*, so that the next time this state shows up in the search we know that we do not have to explore it again.

If the size of a single state descriptor for this example is 2^{10} bytes, we would not be able to store more than $2^{35}/2^{10} = 2^{25}$ unique states in 32 GB (about 33.5 million). The bitstate storage method on the other hand, can in principle store up to 2^{38} or 275 billion unique states, for an increase of almost four orders of magnitude.

In sufficiently many bits are used to index the hash-arena, the probability of hash collisions can be very small. A hash collision would happen if two states that are not equal produce the same 35-bit hash signature. The search truncates in this case, which means that it is no longer guaranteed to be exhaustive. We can reduce the probability of hash collisions effectively by using multiple hash-functions for each state, which then corresponds to checking and setting multiple bit-positions. If h hash-functions are used, for instance, a hash-collision must now occur on all h bit-positions for the state to be affected. The Spin model checker uses three hash-functions per state by default, but the number is user-selectable.

For large state spaces, or large state descriptors, where exhaustive storage of a complete statespace with a traditional method is infeasible, the bitstate storage method is an attractive alternative to increase problem coverage, by a significant margin. Another advantage of the use of a bistate storage method for large problem sizes is that for a given size of the hash-arena, the maximal runtime of the algorithm is fixed and known, independent of the actual statespace size.

References

1. B.H. Bloom, "Space/Time Trade-offs in Hash Coding with Allowable Errors", Comm. of the ACM, Vol. 13, No. 7, pp. 422–426 (1970).
2. D.L. Detlefs, C.H. Flood, A.T. Garthwaite, et al.: Even better DCAS-based concurrent deques. In: Maurice Herlihy (ed.) Distributed Algorithms, LNCS, vol. 1914, Springer, Heidelberg, pp. 59–73 (2000).
3. S. Doherty, D.L. Detlefs, L. Groves, et al.: DCAS is not a silver bullet for nonblocking algorithm design. In: P.B. Gibbons and M. Adler, (eds.) Proc. Sixteenth Annual ACM Symposium on Parallel Algorithms, Barcelona, pp. 216–224 (2004).
4. Dwyer, M.B., Elbaum, S.G., et al.: Parallel Randomized State-Space Search, Proc. ICSE 2007, pp. 3–12 (2007).
5. Havelund, K.: Mechanical Verification of a Garbage Collector, Proc. Int. Workshop on High-Level Parallel Programming Models and Supportive Environments, Puerto Rico, pp. 1258–1283 (1999).
6. Holzmann, G.J.: On limits and possibilities of automated protocol analysis, Proc. 6th Int. Conf. on Protocol Specification, Testing and Verification INWG IFIP, pp. 339–344 (1987).
7. Holzmann, G.J., Smith, M.H.: Automating software feature verification, Bell Labs Technical Journal, Vol. 5, No. 2, Special Issue on Software Complexity, pp. 72–87 (2000).
8. Holzmann, G.J.: The Spin Model Checker – Primer and Reference Manual. Addison-Wesley, Mass. (2004).

9. Holzmann, G.J., Joshi, R., Groce, A.: Swarm Verification Techniques. IEEE Trans. on Software Engineering, Vol. 37, No. 6, pp. 845–857 (2011).
10. Holzmann, G.J., Florian, M.: Model checking with bounded context switching, Formal Aspects of Computing, Vol. 23, Issue 3, pp. 365–389 (2001).
11. Holzmann, G.J.: Mars Code. Communications of the ACM, Vol. 57, No. 2, pp. 64–73 (2014).
12. Erratum to [11], http://spinroot.com/dcas/
13. Joshi, R., and Holzmann, G.J.: A mini challenge: build a verifiable filesystem, Proc. Verified Software: Theories, Tools, Experiments, Formal Aspects of Computing, Vol. 19, No 2, pp. 269–272 (2007).
14. Modex, a model extractor for C code, http://spinroot.com/modex/
15. Penix, J., Visser, W., Pasareanu, C., et al.: Verifying Time Partitioning in the DEOS Scheduling Kernel, Formal Methods in Systems Design Journal, Volume 26, Issue 2, pp 103-135 (2005).
16. The logic model checker Spin, http://spinroot.com/
17. The Fleet distributed system, https://github.com/coreos/fleet/