Using Save/Restore is Easy, Right? A User's Perspective on Deploying Save/Restore in a Mature Verification Methodology

Ed Powell (ed.powell@hpe.com), Ron Thurgood (ron.thurgood@hpe.com),
Aneesh Samudrala (aneesh.samudrala@hpe.com)
Hewlett Packard Enterprise, 3404 E Harmony Road, Fort Collins, CO 80528

***Abstract*-ASIC designs today are growing in size. The use of interposers and multiple chiplets creates even larger system topologies that need to be simulated. As simulation topologies grow, the simulation performance suffers greatly, leading to very long test times. In some large designs, just the initialization step of a system simulation can take hours to days. In many cases, the initialization sequence does not perform anything unique, making the running of this phase of the simulation necessary but not valuable and leaving less time available to focus on the interesting aspects of the simulation. These factors are driving the need to integrate new approaches, such as save/restore, into the central verification methodology. Our team has fully integrated the save/restore technology into our verification methodology and this paper will explore how we accomplished this integration, the challenges we faced, and how we solved those challenges. Finally, the real benefits we have observed from this integration will be presented.**

I. Introduction

For many years, the concept of saving a snapshot of a simulation, restoring the saved state later, and continuing the simulation has been suggested as the basis for improved verification productivity. Until recently, practical limitations of vendor implementations for this feature have prevented broad adoption of save/restore in our methodology. We have found that the vendor solutions are now mature enough to handle our sophisticated simulation environments. However, having a functional technology solution does not create a methodology. Our team has taken the save/restore technology solution and developed a robust methodology to extract the full advantage of this capability across our entire project development efforts. While the concept of save/restore is not new, understanding how to practically integrate it into a mature and automated verification methodology is difficult. We will present how we accomplished this integration and showcase the real results we have seen from it.

II. Background

From the earliest versions of Verilog simulators, starting with Verilog-XL, the concept of save and restart have been present. The basic idea is that at some point in a simulation the current state of the design is written to a file. This state can then be re-loaded and the simulation continued from that point. If we can run a simulation through the initial setup and save that state, and then restore and run many simulations from that point, then we can save lots of time by avoiding re-simulating the common initial sequence in each test. There have been many challenges encountered with implementing this basic idea through the years. For example, the initial attempts saved only the state of the Verilog model itself. User code written in C and included with the simulation using PLI/VPI needed to implement save and restart handlers to capture and restore the state of the user code. Attempts to write handlers for this proved to be very difficult. A large complex behavioral C model can incorporate code from a variety of sources and it is very hard to account for every last state variable used in a model. If the simulation uses third party PLI/VPI models, they would often not implement save/restart handlers. There was also a challenge with supporting mixed language simulation such as with VHDL models. More recently the save state approach has been based on saving and restoring of the whole simulation process image, which avoids requiring each language and model to provide a save/restart handler.

A. Past Solutions

The original mechanism provided by Verilog-XL, and included in the IEEE 1364-1995 specification, was implemented as the `$save()` and `$restart()` system tasks. These system tasks could be called from Verilog code implementing a testbench. Typically, a simulator could also execute these operations from a TCL command prompt. User or third party C models were required to implement a Program Language Interface (PLI) `misctf` callback `reason_save` or `reason_startofsave` function that would be triggered when `$save()` was called. This process prepared a block of memory representing the state of each model instance to be included with the save file. When the `$restore()` was called the model was required to have implemented callbacks for `reason_restart` or `reason_startofrestart` to read the saved block of memory and use it to restore internal model variables to the original state for each model instance.

With IEEE 1364-2005 the PLI was deprecated and replaced with the Verilog Procedural Interface (VPI). The `$save()` and `$restore()` system tasks were carried forward, and the method of saving state for a model now used the VPI callback reason `cbStartOfSave`, `cbEndOfSave`, `cbStartOfRestart`, `cbEndOfRestart`, with the `vpi_put_data()` and `vpi_get_data()` functions.

With IEEE 1800-2009 the IEEE-1365-2005 Verilog specification was merged with the SystemVerilog specification IEEE 1800-2005. The VPI support for save/restart system tasks and callbacks continued until the current SystemVerilog IEEE 1800-2017. In this traditional save and restart approach, the task of capturing state for C models is up to the model writer. Over time, improvements were made, such as support for mixed language HDL models, but with large verification environments using Specman/e for verification code, it was still not possible to use save and restore.

To address the difficultly of writing the save and restart callback routines and to support verification code such as Specman, simulation vendors have started to provide an alternate approach based on saving the entire process state.

B. *Current Solutions*

Our simulation environment has long supported different ways to run simulations, with the ability to share parts of the execution flow to get better efficiency. Figure 1 shows two *run modes* that we have supported. The first doesn't share anything between tests. Each test runs through the entire simulation flow. The second has the elaboration step shared between all tests running on that topology. Each test loads the *topology elaboration snapshot*, and begins running the test. This mode reduces the number of elaboration jobs run, saving run time and license usage. Over a large regression, the savings can be quite large. For our larger simulations, we are saving over an hour of simulation time per test by sharing the topology elaboration snapshot. We have wanted to deploy save/restore later in the simulation cycle but due to limitations in the technology, and complexities in our environment, this was not feasible; however, simulators are now offering a save and restore capability based on saving the entire process image. Once the entire simulation process memory state is being saved and restored, we no longer have to worry about save and restore for individual C or SystemC models, other modeling languages, or verification code. With this approach, we are now able to support moving the save/restore technology later in the simulation cycle. Figure 2 depicts how we can now have tests share the "init phase" of a simulation with multiple tests, by creating an *init snapshot.* Our large topologies can now avoid hours of simulation cycles per test from the init phase in addition to that saved from sharing the elaboration time.

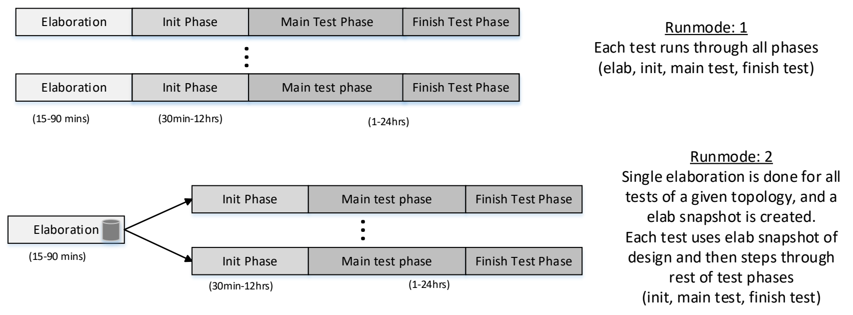


Figure 2: Simulation Run Modes

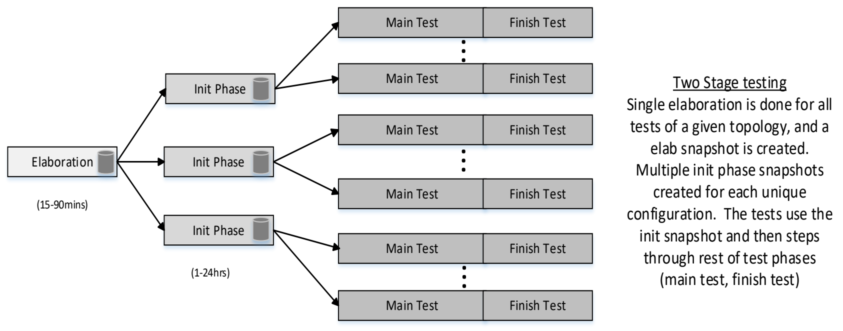

Elaboration
(15-90mins)
Init Phase
Init Phase
Init Phase
(1-24hrs)
Main Test
Finish Test
Two Stage testing
Single elaboration is done for all tests of a given topology, and a elab snapshot is created.
Multiple init phase snapshots created for each unique configuration. The tests use the init snapshot and then steps through rest of test phases
(main test, finish test)

Figure 1: Two Stage Testing

With this new approach there are still other challenges. For a restarted simulation to be useful, we need to be able to do more than just continue running the original simulation exactly as it was started. The goal of improved overall verification efficiency requires that we run many different tests starting from the same init snapshot. This means that we need a way to dynamically change the random seed used, adjust the test code to execute, and possibly tweak the waveform dumping and logging options after loading the init snapshot. There are also environment variable settings that must be adjusted to reflect the restarted jobs' run environment. All of these are part of the saved process state and must be adjusted after loading of the snapshot.

Most recently our simulation tool now supports save and restore of the process state when C code in the simulation includes use of pthreads. Some of the third party models we currently use fall into this category. Now with this feature supported, we have been enabled to apply save and restore to our large system simulations that include mixed VHDL and Verilog models, Specman/e and SystemVerilog verification code, and third party C bus functional models (BFMs). We are able to set a new random seed for SystemVerilog and Specman/e and we are able to dynamically load Specman/e test code after loading an init snapshot.

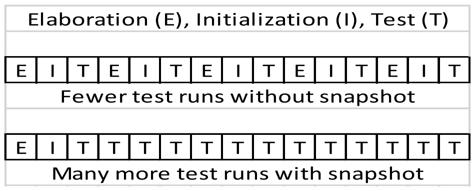
III. Integration Into a Methodology

A. *Productivity Improvement Areas*

Given that we now have a viable save/restore capability provided by our simulation tool vendor, our challenge was determining how to effectively deploy use of it in our highly automated constrained random verification environment and flows. We wanted to utilize save/restore to improve productivity in three basic areas: simulation throughput, test development, and debugging.

i. *Improved regression testing throughput*

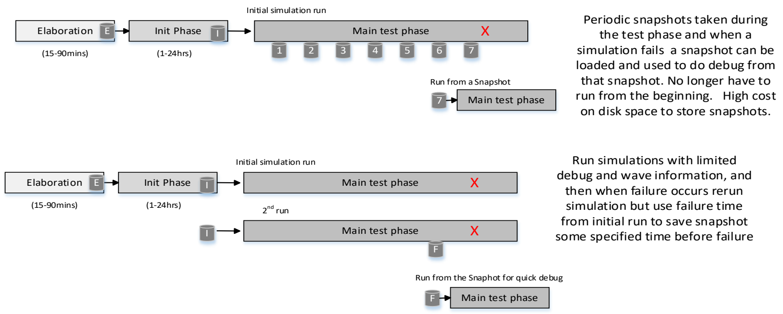
One goal was to reduce the re-running of the same initialization portion of the simulation over and over. By running many different tests starting from the same *init snapshot*, we focused regression testing on the "interesting" part of the simulation rather than wasting time re-running the same initialization with each test. See Figure 3, where "E", "I" represents the running of the initialization phase, and "T" represents the running of the main test phase of the simulation.


Elaboration (E), Initialization (I), Test (T)
E I T E I T E I T E I T E I T
Fewer test runs without snapshot
E I T T T T T T T T T T T T T
Many more test runs with snapshot
Figure 3: Init Snapshot Provides Greater Throughput

ii. *Efficient regression test failure debug*

By saving a snapshot just before the test failure, the engineer debugging a failure is aided greatly since they no longer have to run a simulation from the start to generate additional debug information. We found two different use models that we deployed to improve user debug efficiency, as shown in Figure 4. Generally, we have tried to keep the test runtime after loading of the init sequence as short as possible, so restarting a test for debug from the init snapshot works well. For debugging efficiency, it is better to have many short tests that cover specific features. However, to support all of our use cases, the methodology needed options to also improve debugging for long running tests.

The first use model periodically saves a snapshot during a long running test. When enabled, this code writes a snapshot every N minutes of wall clock time. In order to save disk space we have capability to indicate the number of snapshots to save (default is 2). If a test failure occurs, the user can then run starting from one of these snapshots and will know they can reproduce the test failure after no more than 2N minutes of sim time, while insuring there is enough simulation before the error to understand what is happening.


Initial simulation run
Elaboration
E
(15-90mins)
Init Phase
I
(1-24hrs)
Main test phase
X
1
2
3
4
5
6
7
Run from a Snapshot
7
Main test phase
Periodic snapshots taken during the test phase and when a simulation fails a snapshot can be loaded and used to do debug from that snapshot. No longer have to run from the beginning. High cost on disk space to store snapshots.
Initial simulation run
Elaboration
E
(15-90mins)
Init Phase
I
(1-24hrs)
Main test phase
X
2nd run
I
Main test phase
X
F
Run from the Snaphot for quick debug
F
Main test phase
Run simulations with limited debug and wave information, and then when failure occurs rerun simulation but use failure time from initial run to save snapshot some specified time before failure
Figure 4: Efficient Regression Test Failure Debug

The second approach optimizes this flow further and relates directly to how we run regressions. The initial test run has limited log information and no wave data captured to optimize simulation performance. Upon a test failing, our regression scripts automatically rerun the failing test with a higher level of verbosity and wave data being captured. We augmented this functionality to have the ability on the rerun of the test to have a snapshot created a specified amount of time prior to the failure. This allows the user to spend less time rerunning to get to the point where a failure occurred.

Prior to the introduction of snapshots, a user would have to test minor changes or fixes by running the entire simulation back from time zero. Now with two stage tests, if a failure occurred after the snapshot was created, a user can simply run from the time the snapshot was created. The user can make quick changes to the files loaded after the snapshot was created and rerun the simulation. With the proper test structure in place, this might only require simple constraint changes. The debugger can be used in a similar fashion where the user can load the design snapshot and save time waiting for the simulation to get to the point of the failure. As a result of all these tools, the test developer is able to iterate through failures efficiently without wasting compute resources and time.

iii. Test development efficiency

For improving test development, we used save/restore to reduce the test development cycle. By loading a saved snapshot and dynamically loading a new version of a test, the developer can save runtime compared with re-running the same initial sequence each time a new version of the test needs to be run. Since our environment utilizes both SystemVerilog and Specman/e for verification, we explored the unique differences between these languages as they relate to a save/restore methodology. Specman/e is the primary test language for all our large topology verification. Cadence offers a unique feature of dynamically loading test files after loading of a snapshot. Dynamic file loads (DFL) are very flexible and allow for users to modify almost all parts of the testing environment. Taking advantage of aspect oriented programming, the user is able to extend previously defined objects and methods. In addition, the user can create new sequences, checks, and TCMs providing endless variations from a single snapshot. With all these features, test developers would be able to write tests on top of a snapshot and not have to wait for the entire design to compile. Entire design compilations can be saved for major testbench changes helping save simulation cycles for test developers. However, a lot of these features currently are not fully functional in our test benches. The test development section below goes more in depth on what features worked and the limitations we encountered.

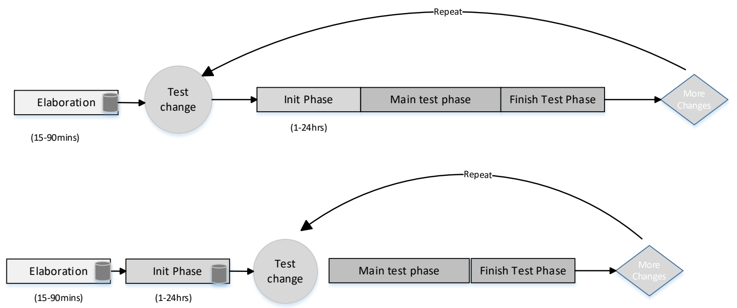


Figure 5: Test Development Efficiency

System Verilog testbenches see benefits from snapshotting as well; however due to the lack of aspect orientation and dynamic code loading, all files must be compiled into the elaboration snapshot, so test development is more difficult. Because System Verilog is not aspect oriented, a specific methodology must be created to allow flexibility of what to run after the init snapshot is loaded. For instance, through the use of virtual sequences, the user has the flexibility of selecting from any of the tests originally compiled into the init snapshot by specifying the sequence name on the command line.

B. Technical Challenges and Complexities

We discovered that when using the save/restore capability beyond very simple use cases, there were additional complexities that had to be addressed to fully deploy the technology and enable the multiple use cases of this technology. Most ASIC development teams utilize some degree of automation within their methodologies and we found save/restore can be integrated into any development methodology; however, there were practical challenges that we had to address.

i. Optimal Job Launching and Dependency Tracking

In our existing two step test launch system, the launcher creates an elaboration job for each unique combination of design topology and options. When the elaboration job finishes, the tests using that topology elaboration snapshot are executed. The challenge we faced with save/restore was to automate the management of init snapshot jobs in a similar manner. The launcher must now keep track of both elaboration snapshots and init snapshots and the dependencies between elaboration jobs, init jobs, and test jobs.

Figure 6 shows this complexity. Several init sequence tests might run after loading a given elaboration snapshot. Each of those initialization tests might be run with a different seed value, compiler defines, or constraints that affect the specific init snapshot that is produced. To keep track of which tests can run from which init snapshot and from which elaboration snapshot, the launcher algorithm computes an elaboration signature for the topology given the base topology directory and options affecting elaboration.

Additionally, for all two stage tests using a given elaboration signature, we determine an init signature based on the init sequence test, constraints, and options that affect running of the init sequence. The launcher sets up job dependencies such that the init jobs wait on the corresponding elaboration job for a given elaboration signature and the test jobs wait on a given init job for a given init signature. This approach ensures we optimize throughput where elaboration and init snapshots may take varied times to be generated.

ii. Multiple Init Seeds

The challenge of initialization randomization must be properly addressed to allow trading off the number of random seeds used against the amount of disk space needed for init snapshots. With our methodology, the user can specify in the test description a repeat count for both init jobs and test jobs, causing the job to be run multiple times. Each repeated init job is run with a different initial seed and, likewise, repeated test jobs are run with different seeds. The repeated tests are distributed across repeated init tasks with compatible signature in a round robin fashion. The launcher tool has the option of saving these elaboration and init snapshots in a common area, and, if the same tests are launched later, the user can specify to reuse the existing init job snapshots. This saves a lot of time when debugging issues in which the design and init sequence have not changed, but we are trying out changes to the dynamically loaded part of the test.

Figure 6 illustrates the complex job hierarchy described above. For the scenario shown, we have four tests (A, B, C, D) run in a simulation topology named X. Tests A and B define a different set of simulation options than Tests C and D, causing two elaboration signatures (Topo Sig1, Topo Sig2) to be created. As a result, two elaboration jobs are executed as shown in the first column of Figure 6. Additionally, let's assume test A calls for "safe" random initialization values to be used and Test B calls for "full" random initialization values to be used. This creates two different init signatures which causes two init jobs to be created. Similarly, Test C calls for "safe" random initialization values to be used and Test D calls for "full" random initialization values to be used. However, since Tests C and D are associated with topology signature 2, two new init signatures are created and separate init jobs are scheduled. This is illustrated in the second column of Figure 6.

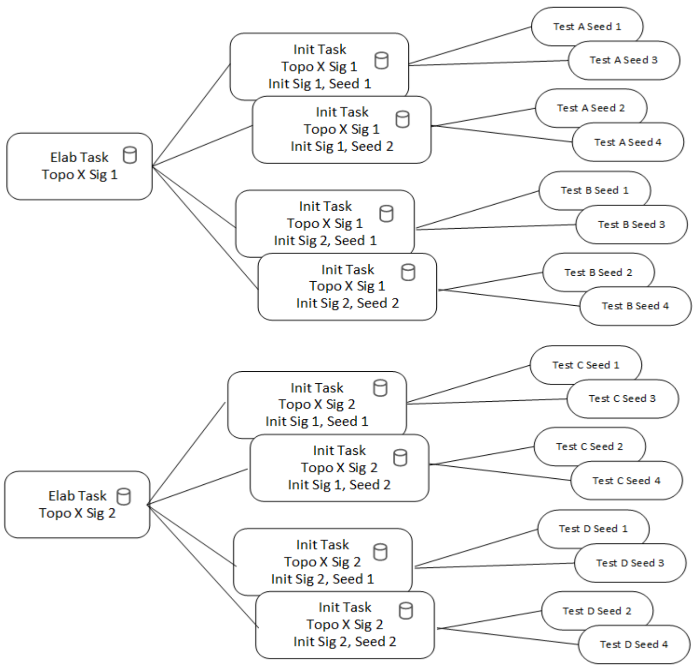


Figure 6: Managing Job Signatures and Init Seeds

When the initialization sequence is randomized, each simulation run would normally create a different random initialization by having a different seed for each test run. Even though we are utilizing save/restore, we do not want to give up using multiple random initializations. We accomplish this by specifying an init repeat count. In Figure 6, the init repeat value was set to a value of two which tells the launcher to schedule twice the number of init jobs, each with a different init seed. This allows our team to realize the benefits of init snapshots while still benefiting from randomization during the init phase. This is easily accomplished by setting the init repeat value to the number of unique seeds wanted.

Finally, the ultimate goal is to run the main test simulations. The tests also are given a repeat value and Figure 6 shows all tests having a repeat count of four, meaning the test is run four times, each with a different seed value. Each repeat of a given test loads an init snapshot and sets a different seed value to use for the simulation run.

Using this launcher dependency structure for jobs and repeat count scaling factors, we can easily scale up or down the number of simulations that are run while having direct control on the number of snapshots generated.

iii. Dynamically Changing Run-Time Settings

Another complexity we encountered when running two-stage tests is with changing the simulator settings for the depth of wave recording for signals, messages, and transactions. Typically, many of these settings are set at the beginning of a simulation. With the use of init snapshots, the beginning of a simulation is reflected in the init snapshot. When you load an init snapshot, the behavior for most settings will be those from the original init sequence run. To allow users to rerun tests with certain settings changed, we had to find ways to execute commands to change the settings after the snapshot is loaded. This is done by saving the desired environment settings in a file and then loading the new settings after loading of the snapshot.

Additionally, we found that some third party VIP did not fully support the save restore methodology. Their VIP did not fully update their environment as new simulations were started after loading an init snapshot leading the VIP to believe they were running on a different compute host than currently being used. In order to fully support save restore technology, VIP must some state in their environment.

iv. Specifying a Two Stage Test

We wanted to make the process of writing or converting a test to use save/restore to be a natural extension of our existing test description format. Figure 7 shows an example of a launcher test description for a two-stage test. In the example, the lines that are specific to writing a two-stage test are highlighted in yellow. The `init_name` line specifies the name of the init sequence to run, The `test_name` specifies the test to run after loading the init snapshot. The `test_ecode` adds an additional constraint to the test task. The seed for the test and the init sequence are both set to random and will be set to two different random values. Users can also rerun a previously launched test and choose to use the existing topology or init snapshot when rerunning the test. Alternately, the user can choose to rerun one or both of the elaboration and init snapshots during the test rerun.

```
:Identifier   genz_2stage_atomic_traffic_1_0
:Repeat 4
:Repeat_init 2
:Class  (  all two_stage_turnon)
:Method   simulation
  (
    test_mode ( e )
    topo_dir   (#THIS#/topo/genz_rsp_ip/e)
    test_dir   (#THIS#/test/genz_rsp_ip/e)
    init_name ( genz_standard_init )
    test_name ( atomic_traffic_test )
    compargs (  #G01_0_COMPARGS# )
    ecode    (#ECFG_VER_1_0#)
    test_ecode
    (
      "extend genz_pkt_s {
              keep
global_cid_pres.reset_soft();
       };"
    )
    memory ( 3G )
    slots ( 1 )
    init_seed( random )
    seed ( random )
  )
:Owner     Ed Powell
:Summary Demonstrate save/restart testing
```

Figure 7: Test Description Using Init Snapshots

v. Disk space management

One of the challenges associated with save/restore technology is the amount of disk space that can be quickly consumed as it is deployed across an entire project. The save/restore technology has compression capabilities to reduce the footprint on disk, but this comes at the expense of the time it takes to compress and uncompress the snapshot.

We addressed this challenge in multiple ways. First, we doubled the allocated disks space for the projects using save/restore compared with the storage capacity we have typically used in the past. This also impacted individual teams using save/restore because they would often be forced to use two different disk volumes for their simulation results instead of just one. We adjusted our regression management scripts to account for the need to ping pong between two disks instead of using just one.

Second, we adjusted our disk cleanup processes to be much more aggressive in deleting data when it was no longer needed. We automated this cleanup of regression data with scripting, giving teams direct control on the number of days of passing tests, failing tests, and snapshot data that was to be saved. However, this does typically mean that fewer days of regression results are kept than our teams are accustom to having.

Finally, we had to actively manage the number and size of the snapshots we generated. We have worked closely with our simulator vendor to reduce the simulation memory consumption. One technique we used was to adjust the garbage collection settings to limit the memory consumption while ensuring the simulation performance was not degraded. All efforts which reduce the simulation memory usage directly reduces the disk footprint of snapshots.

vi. Coverage Management

Properly managing coverage results produced from simulations run from a restored state was also considered in our methodology. Fortunately, the coverage state of the design is included with the saved process snapshot, so when we load a snapshot and run a test, the coverage data written out at the end represents all of the coverage from both the init and test stages of the run. As long as we insure the coverage option settings are the same for the init and test runs, then we get the same coverage results as if the test had been run as a single simulation.

C. Test Methodology Challenges

As we deployed the save/restore technology we realized that we were going to have some challenges associated with how our tests were structured in the past. We needed a clean separation from what is being done in the initialization part of the test and what was being done in the main test part of the simulation. Things that could be shared needed to move into the initialization phase so they could be included in the snapshot, whereas things that changed for a test needed to be executed after the snapshot was loaded. We also needed a new methodology around management of the snapshots and management of the required disk space to store the snapshots.

i. Test Initialization Randomization

One of the challenges associated with snapshots is finding a balance on the number of snapshots you have the disk capacity to store and ensuring the team is getting enough random testing in the initial setup of the device under test. In our original methodology, all register state initialization was controlled through randomization at the beginning of the test. Using snapshots means that each test would be using the same initialization sequence, where in the past a user could have some additional randomization for the initial setup of the device in each test. We addressed this in a couple ways. First, as mentioned before, we added capability to have multiple init snapshots created on the same set of tests, allowing for more randomization during initialization. This, however, increases the disk footprint to store additional snapshots. Second, we also addressed this by moving away from using constraints at time zero to using main test phase sequences to initialize any registers after the snapshot has been created. This allow a degree of initialization settings randomization to occur in every test that runs after loading an init snapshot.

In some case, we had specific focused testing where it didn't make sense to create an init snapshot, because the test was more focused on what was happening in the initialization phase or the test was so specific that there was no need for sharing between tests. So the test writer must understand the intent of their tests and determine if using two stage testing will add value.

ii. Test Development

Testing environments rely heavily on randomization and most of the initialization randomization related to testbench setup is done with constraints evaluated at simulation time zero. The problem with time zero randomization when using init snapshots is that the fields and stimulus are not regenerated (i.e. re-randomized) for future tests that run using the same snapshot. To adapt to this limitation, we have created a methodology to move all required stimulus into sequences that can be generated on the fly. That way the constraints get evaluated when the sequence is started.

One challenge we found is that in some cases the user must have knowledge of the order in which to generate fields. Despite this complication there are a lot of benefits that came out of dynamically generating stimulus. A user can add new constraints loaded after the snapshot has been created giving a lot more flexibility when writing tests. The key concept to keep in mind when creating stimulus is to consider how can one extend this later and create tests that can run using the same snapshot.

There are some limitations with test language features that can be used when dynamically loading files in Specman/e. The testbench structure, and anything generated at time zero, cannot be changed by dynamically loaded e code. By adding constraints and extending objects, we are able to write effective dynamic tests. However, to date, we have not been able to successfully extend sequence methods, create new sequence methods, or create new TCMs as we originally expected. We are working with our simulator vendor to address these limitations.

In order to fully utilize the benefits of two stage testing, our test methodology had to be modified. Current limitations do not allow us to modify the body of a sequence after time zero. To work around this issue, a user must create their test sequence, have it compiled in as part of the snapshot, and use a master sequencer to decide what sequences the test runs after the snapshot is loaded. We have created a simple user experience through the use of macros and foundation code that allows a user to specify exactly what they want to run. This methodology takes advantage of on-the-fly generation and constraints that can be loaded after a snapshot is loaded.

D. User Experience Challenges

Two stage tests have changed the testing environment drastically for users. Users have many new variables to keep in mind, such as disk space, snapshot reusability, and simulation cycles. With huge system level simulations, save/restore allows higher throughput on various types of tests, helping to catch more corner cases that could have otherwise been missed. The downside to this approach is that regression will run the system initialization with fewer unique seeds and less randomization of DUT initialization settings. In addition, if the initialization fails, then no tests will run that night due to their dependency on the init snapshot step. This results in no test throughput for those tests. However, with proper planning this can easily be avoided.

Overall, snapshotting a simulation has a lot of benefits but requires some ground work. The test bench owner needs to keep in mind the requirements associated with snapshots and design the test bench accordingly, with an easy way to change stimulus by providing the proper knobs to tweak the test after initialization. Also the test bench designer should figure out the optimal time to create a snapshot. Too far into the simulation will cause less flexibility and too early into the simulation will cause tests to be much longer. As many tests as possible should be able to run with a given snapshot to minimize how many snapshots are needed.

IV. Results

Significant improvements in productivity have been achieved for large designs by using the save/restore technology and it has enabled our team to utilize system level simulation environments that were previously simply too large and slow to be practical. Ultimately, with the technology fully integrated into our methodology, our teams are enabled to quickly and easily utilize this capability and realize its value with no additional development on their own.

We have seen real benefits to our team with save/restore technology integrated into our core verification methodology, including improvements in regression simulation throughput and more effective use of expensive simulation licenses. With this methodology deployed on actual projects, we have also seen engineering benefits in areas such as debugging and test development. While test development approaches must change to maximize the benefit of this technology, we have found the benefit far outweighs the cost.

A. *Simulation Throughput savings*

The basic benefit of using save/restore is to skip the time needed to run the same initialization cycles over and over for each simulation. This was illustrated simply in Figure 3 above. Here, we will provide actual results from using save/restore on our current project.

These results were collected from a configurable chip level simulation environment. By using configurable parameters, we were able to adjust the size of the DUT, resulting in simulations requiring different amounts of simulation memory, different init snapshot sizes, and different simulation performance characteristics. Table 1 describes the size of the three different topology configurations we are summarizing.

Table 1: Disk Usage from Regression

Topology Size	Design Size: Module Instances	Design Size: Register Instances	Simulation Run-Time Memory Usage (GB)	Typical Simulation Duration (hrs)	Typical % of Duration Spent in Init Phase
Medium	100-200K	1-2M	15-20	0.3	40-50%
Large	650-1600K	7-11M	50-60	1.0	50-60%
Extra Large	2-10M	15-45M	100+	2.5	40-50%

One of the first areas we looked for data is the time required to save and load an init snapshot. This is an important portion of the ROI calculation because if the save and load times are long, this will quickly take away from the benefit of the save/restore methodology. What we found is that with today's technology, the save and restore times are negligible compared with the init phase savings. The larger simulation topologies did take longer to save and load the snapshots but all were in the range of 1-4 minutes. In the next section we will also share our findings regarding tradeoffs between save and load times versus init snapshot compression settings.

Our overall savings from using the save/restore methodology is significant. To assess the ROI, we looked at the following items: duration of the init phase of the simulation, save and load times of the init snapshots, and the duration of the main test phase of our simulations. Figure 8 shows we saved more than 50% of our simulation cycles by utilizing save/restore capabilities for a typical regression run. The cycles highlighted in blue show cycles that are necessary to initialize the chip we are validating but that bring no material verification value to the team as this portion is largely the same from simulation to simulation. Figure 8 also shows the negligible portion of the simulation that is devoted to the saving and loading of the init snapshot. As expected, the percentage of this save/load time is greater in smaller topologies with shorter overall simulation run times. When using save/restore, the cycles that had previously been spent repeating the initialization cycles of the simulation are now spent running new cycles of the Main Test phase where the significant verification activity is happening (shown in green in Figure 8). The value directly shows up as significantly more simulation cycles spent exploring new random stimulus scenarios for a given period of time.

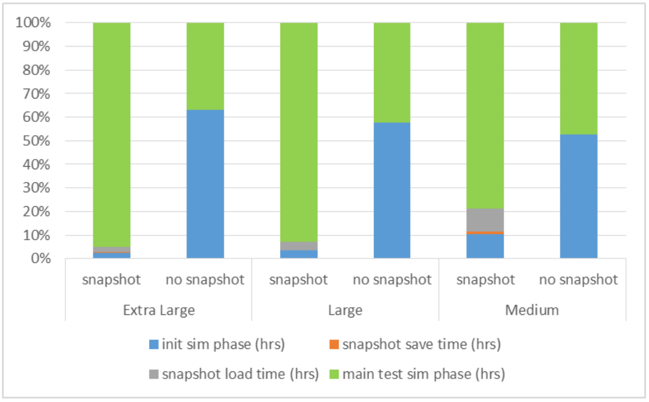


Figure 8: ROI of Save/Restore Methodology

B. *Snapshot and Disk Space usage*

This methodology was being developed by the chip/multi-chip team to support very large simulation that took hours to get through elaboration and initialization. As a result of this, the deployment of the capability has been mainly targeted at topologies above the block level. As we deployed the save/restore technology, we found that we may need to utilize it differently for different environment sizes. The teams must balance the number of snapshots and their total storage size to ensure adequate resources are available.

TABLE 2: DISK USAGE FROM REGRESSION

TOPOLOGY SIZE	AVERAGE INIT SNAPSHOT DISK SIZE	AVERAGE INIT SNAPSHOTS CAPTURED PER DAY	TOTAL DISK SPACE USED FOR INIT SNAPSHOTS PER DAY
Medium	900 MB	27	22.5 GB
Large	1.6 GB	26	41.6 GB
Extra Large	3.6 GB	35	126.0 GB

Table 2 captures information from our actual daily regression runs, which were restarted daily with the latest DUT model available. Many of our block teams typically share a 500 GB disk for their regression results. Table 2 shows that such a small allocation of disk space would not be sufficient for these larger simulations using init snapshots. Since for passing tests we save very little data, for our largest topologies we consistently see 50-60% of our regression disk space consumed by the snapshot data when the regression run has a low failure rate.

To manage disk space we rotate results across multiple disks, and aggressively clean up snapshots and results once they are not needed. We also use compression of the snapshot to reduce disk space needs. As shown in Table 3, compression adds a little extra time to save and load the snapshot, but the disk space saved is well worth it.

TABLE 3: IMPACT OF ENABLING SNAPSHOT COMPRESSION

TOPOLOGY SIZE	COMPRESSION SETTING	SIZE OF INIT SNAPSHOT (GB)	REDUCTION IN SNAPSHOT SIZE (%)	TIME TO SAVE SNAPSHOT (SEC)	TIME TO LOAD SNAPSHOT (SEC)
Medium	None	6.5	-	16	27
	1	1.2	81.5	70	51
	3	1.1	83	80	50
	5	1.1	83	116	49
	7	1.09	83.2	242	49
Large	None	12	-	30	42
	1	1.9	84	125	88
	3	1.8	85	151	90
	5	1.8	85	205	88
	7	1.7	85.8	419	88
Extra Large	None	23	-	55	101
	1	3.6	84	209	154

V. RECOMMENDATIONS

Our results show significant improvement in overall simulation throughput and in user debug productivity for large system level designs. In fact, without deployment of the save/restore methodology for our current system design project, it would have taken many more resources to complete our project on time. If your project has large designs that are taking a long time to simulate, and a significant portion of the simulation time is devoted to standard startup operations such as register initialization and link training, then developing a save/restore methodology will likely be of benefit.