

Software debug is a highly iterative process. Execute the system model, hit a break-point, step through the code, set a different break-point, re-run the model, etc. Once the problem has been found, the software is modified and recompiled and execution starts again. The changes, however, are usually made to code that is executed after some amount of execution and rarely in the very beginning.

When changes are made to the software that is being executed, the initial part of the software is usually not changing and you may execute quite a bit of code before your changes actually impact the simulation. A change in the software that occurs after an initialization phase, for example, might not cause any changes in the system behavior for thousands of cycles. This means that a significant amount of the execution being performed after a minor change is often actually a complete duplication of the effort that was already performed during the previous execution. For example, if there is a change to a software routine that does not get called until after 10ns, the behavior of the system will be the same for the first 10ns.

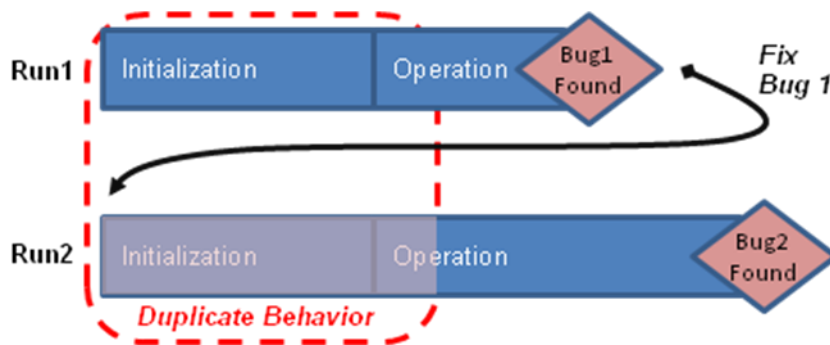


Fig. 1 – Duplicate behavior after changes

Save and restore

Ideally, we would like to be able to pick up the execution from the point at which the code changes begin to make an impact. In our example before, this means that we would like to make the code changes and re-start the execution at 10ns, instead of re-running the first 10 ns of activity over again.

This presents several problems in practical cases. There may be black-box or other types of IP in the system model that will not allow us to perform a save and restore operation on them. Even if we can overcome these obstacles, how do we really know at what point the results diverge from the previous run? If we guess too late then we will have incorrect results because we will restore the system past the point at which the changes actually start to impact it, If we guess too soon, we do not take advantage of the performance benefits of reusing all the results that are the same.

Another issue is that changes might show up in some parts of the system earlier than others. If we make a change to the memory initialization code, this change might impact the CPU and memory units right away but not cause any change in the behavior of the Jpeg decoder until much later – or even at all. In this case we would like to be able to save and restore different parts of the system at different times.

Solution

The ideal solution is to be able to restart each major block in the system only at the point at which their behavior is going to change from the last run. Obviously this doesn't work because as soon as any block starts to execute, it will require interaction with the other blocks and will need signals from those blocks and will be sending signals to those blocks.

A way of solving this problem is to capture the behavior of each block at the I/O boundary and re-use these captured I/O signals instead of actually computing the behavior of that block. If we re-use the stored signals at the I/O boundary, we can save the computation that would have been required to re-compute the behavior of that block. This reduces the required computation and boosts performance.

When we run into the part of the execution where the changes are starting to affect that particular block, we start executing again and stop using stored signal values.

How do we know when the behavior has changed? In addition to re-using the stored output values of the block, we have also stored the input signals. For each cycle, we can compare the actual input signal values with the stored input signal values. As long as the stored and actual values match, we know that the output will be the same as from the previous run. This is exactly the problem that Replay addresses.

Let's take a look at how this works in some more detail for a particular block.

First, we execute the system model and store the signal transitions at the primary I/O boundary – both input and output

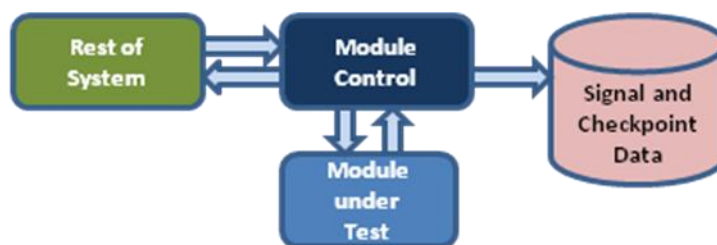


Fig. 2 – Capturing signals

Next, we assume that we have made a change to the software or some other section of the design and we can then replay the captured behavior of the block we are looking at. Because the logic for this block is the same, as long as the inputs match our stored values then so will the outputs. We re-execute the model and at each cycle we replay the stored output values, and compare the input signals with the stored input values.

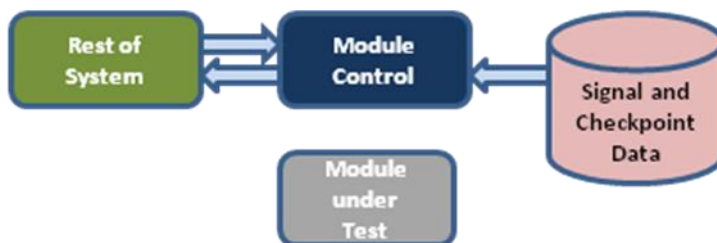


Fig. 3 – Replaying signals

At some point, the input signals will not match the stored values and we will know that we will need to start simulating. Of course we will also need to restore the block to the current values. While we could have been storing the value of each register element at each cycle along with the I/O signals, it will be much more efficient to perform a periodic checkpoint instead of one at every cycle. While this means that we will need to “back-up” in our execution for that block, it will only have to be as far as the last checkpoint.

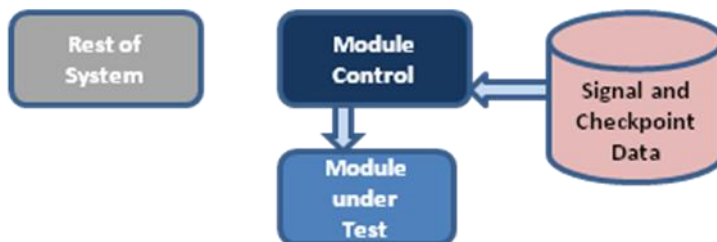


Fig. 4 – Restoring from a checkpoint

Once the block has been brought up to current-time, execution will continue as normal until stopped.

Replay

The effectiveness of this technique relies upon our ability to speed up our model’s execution by assuming that the time to perform the reads and write of the signals (and checkpoints) is less than the time to perform the actual execution of that block. Obviously, a large block with lots of internal activity and a small number of I/O pins will show a more dramatic speed-up than a small block with many I/O signals. In the former case, the cost of storing and replaying the I/O signals is small when compared to the

effort required to actually executing the block's behavior. In the latter case, at some point the cost of storing and replaying the signal data will start to exceed the effort to execute the block.

One easy way to help reduce the amount of data that must be stored is to sample the I/O signals only on the clock cycles. Because the Carbonized models are already using a cycle-level interface, we insure that we only need to sample or analyze the signal values once each cycle, instead of at random times during each cycle.

The key functions that Replay provides include:

- Monitor and capture the inputs and outputs for a specified block
- Checkpoint all of the register values at given intervals
- Replay signals from a stored file to use as output values
- Compare stored inputs with the actual inputs
- On mismatch, load the most recent checkpoint values, allow the block (and only that block) to execute until the current time, and then continue with normal execution from that point

Using Replay

Replay is easy to use and there are only a few controls to set. The user is provided with access to several simple parameters to control Replay during execution. These include the following:

- The name of the activity data file
- The number of cycles before taking the first snapshot
- Recover percentage (checkpoint decay)

Since there is a penalty associated with each checkpoint (stop the execution, dump the checkpoint data, and continue) we do not want to checkpoint on every cycle. On the other hand, if we checkpoint too infrequently we negate the value of the replay because we have to go too far back to rerun the execution for that model.

For example, if we take a checkpoint at 500 cycles when we record our first run, and on the re-run we hit a divergence at 900 cycles, we will have to restore the module's state at 500 cycles and re-run the execution of that module for 400 cycles to "catch-up" to the rest of the system model. This saves us having to run the first 500 cycles, but means that we have to re-run 400 cycles worth of activity. If we checkpoint instead at every 100 cycles, we would only have to re-run 100 cycles worth of activity but then would have the overhead of saving 9 checkpoints in this example instead of one.

A value for how many cycles to wait until the next checkpoint should ideally be based on how long the model execution takes. If the model is only running for 1000 cycles, then

check-pointing at every 100 cycles might be a good first guess. If we are running for 10,000 cycles, then check-pointing at every 1,000 cycles might be a good option. Since our goal is to make this process automatic and not require the user to make manual calculations, the best solution is to use a “decay” in the checkpoint interval so that as the execution continues longer, the interval between checkpoints becomes longer.

Replay allows the user to set the interval between start and the first checkpoint, and then a recovery percentage to indicate what percent of the total execution will need to be repeated (max) after a divergence. For example, if we set the recovery percentage at 10% and the execution hits a divergence at 200 cycles, we would expect to have to rerun a maximum of 20 cycles worth of activity to sync the module with the rest of the system model. If the divergence comes at 500 cycles, we would expect to require no more than 50 cycles to sync the module.

This gives us the best compromise between minimizing the number of checkpoints and also minimizing the amount of time required to sync the module after a divergence.

Conclusion

Replay helps you debug systems faster and is perfectly suited to the repetitious cycles of examination utilized during software debug. Instead of having to re-execute the entire system behavior each time the model is executed, Replay will let you leverage system activity captured in previous executions. Long initialization cycles and other set-up activity that does not change from one run to another can be effectively skipped, allowing the system execution to be limited to the code that is being examined or modified.

John Willoughby is Director of Marketing at **Carbon Design Systems**, based in Acton, Mass.