Transparently Checkpointing Software Test Benches to Improve Productivity of SoC Verification in an Emulation Environment

Ankit Garg[1] K. Suresh[1] Gene Cooperman[2] Rohan Garg[2] Jeff Evans[3]

[1]Mentor, A Siemens Business
Noida/India
{ankit_garg, k_suresh}@mentor.com

[2]Northeastern University
Boston, MA
{gene, rohgarg}@ccs.neu.edu

[3]Mentor, A Siemens Business
Austin, TX
jeff_evans@mentor.com

Abstract*-Traditionally hardware emulation has been used in in-circuit emulation (ICE) mode where the design under test (DUT) executes inside the emulator and connected to the real target, which acts as a testbench. Over time, the software testbench-based emulation environments have become very popular, since the users can control its operation remotely from their desktops. This makes emulators an enterprise resource, accessible to a multitude of users spread across continents and multiple time zones. And since emulators are expensive resources, it is important to utilize them efficiently. Full checkpoint save/restore capability of emulation jobs helps the utilization by enabling flexible job scheduling, shortening of jobs by jumping ahead to interesting points for debug, carrying out what-if analysis, etc. Emulators have native save/restore capabilities for the model on the emulator. The software testbenches can be complex in multiple dimensions. For example, they may be using C/C++, SystemC, SystemVerilog, etc. They may be multi-threaded and based on multiple processes, they may be using IPC, and so on. It then becomes a challenge to save the states of such sophisticated software testbenches both transparently and through a uniform, reliable, mechanism. Since the objective is to solve the problem at an enterprise level, it is critical to find a uniform solution for a diverse set of software testbenches throughout the enterprise. The DMTCP (Distributed MultiThreaded Checkpointing) package supports such a uniform solution. This paper describes the integration of DMTCP with a virtual testbench-based emulation. This brings large benefits to a real life environment that includes multiple emulators within the larger set of enterprise resources. There are other, additional applications of full checkpoint save/restore for emulation jobs that become apparent only after having gained experience with its use in job management. For example, after having observed the behavior of an application prior to checkpoint, additional triggers can be inserted at the time of restore, to enhance debugging of an exception or other unusual behavior.

I. Introduction

The increasing complexity of SoCs, combined with the increasing pressures of time to market, have necessitated efforts to make verification more efficient as well as more effective. On one hand, this has forced execution-based verification paradigms (one of the primary paradigms) to move more and more from simulation to emulation. On the other hand, it has also necessitated efforts to make emulator utilization very efficient and effective.

This is also one of the reasons why emulation with virtual or software testbenches has become popular and it's one of the primary modes of emulation nowadays. This is because the software-only nature of the test-environment enables use of emulators as enterprise resources, shared across a multitude of verification users. Also, enterprise distribution systems managing the access to emulators are put into place to ensure efficient utilization of this expensive resource. The distribution of emulation jobs has its own challenges.

Virtual emulation jobs are mostly non-pre-emptive and cannot be stopped in between when high-priority jobs arrive. In addition, a second complication is that, although resources on an emulator can be partitioned to run multiple jobs in parallel, this flexibility comes with its own constraints. For example, most emulators will have a minimum atomic partition size and emulators may only allow jobs sized as integral multiples of that unit. This introduces a corresponding complexity in job scheduling, since jobs may have varying sizes that do not directly match the capacity constraints on the emulator.

The non-pre-emptive nature of emulation jobs can be rectified by using a transparent checkpoint-restart capability for emulation jobs, assuming it can be built. It can provide powerful capabilities such as: restarting after days of running; powerful job management policies like pre-empting low-priority jobs by an incoming high-priority job; fault tolerance through automatic job recovery; and restoration from a checkpoint after a standard initialization or boot sequence. Technology for "checkpointing" the hardware state of an emulator has been available for many years. However, hybrid checkpointing that includes the state of an arbitrary software testbench has been an elusive problem. This is mainly due to virtual testbench environments having characteristics of being non-deterministic, multi-threaded, multi-process, and written in multiple languages like C/SystemC, SystemVerilog, etc.

Attempts have been made to solve this problem based on application-level checkpointing. The software testbench writer is responsible for maintaining the checkpoint restore capabilities in the software stack. This is cumbersome and difficult to maintain as the software testbench evolves. This is further complicated due to multiple languages and possible third-party components involved in testbenches.

DMTCP (Distributed MultiThreaded Checkpointing) [1] is a compelling new technology that provides a capability to perform binary-level checkpointing completely transparent to the application. It can handle distributed, multi-threaded applications "out-of-the-box" in user space, requiring no system privileges to operate. Since it works at the binary level, no changes are required to the user code. DMTCP automatically accounts for fork, exec, ssh, mutexes/semaphores, TCP/IP sockets, UNIX domain sockets, pipes, ptys (pseudo-terminals), terminal modes, ownership of controlling terminals, signal handlers, open file descriptors, shared memory, parent-child process relationships, pid virtualization, and other operating system artifacts. By emphasizing an unprivileged, user-space approach, compatibility is maintained across Linux kernels. Since DMTCP is unprivileged and does not require special kernel modules or kernel patches, DMTCP can be incorporated and distributed as a checkpoint-restart module with-in some larger package.

The present work proposes an integration of the emulation environment with DMTCP as a solution to the emulation job checkpointing problem and discusses the application and results of such an integration.

The rest of the paper is organized as follows: we describe the background in Section II. Section III describes important use cases for the framework described here. Section IV then describes details of the framework for using DMTCP in Checkpoint/Restore for Co-emulation. Section V then describes a case study for skipping the OS boot in SoC validation environment in an *OEM company*. Section VI then describes future work, while Section VII presents our conclusions.

II. Background: Co-Modeling and DMTCP

A. *Review of Co-Modeling Architecture*

Co-emulation, or (transaction-level) Co-Modeling, is the process of modeling cycle-accurate synthesizable hardware models (DUTs) running on an emulator, communicating with testbenches at transaction level via a high-speed link between the emulator and the host system. The reusable testbenches are interfaced to synthesizable transactors co-located with the DUT in the emulator. These "accelerated" transactors convert high-level transactions to signal-level stimuli to drive the DUT. During an emulation run, the hardware communicates with the software testbench using a high-speed link. For every DUT clock or for each time point in the model execution, communication may be required by one or more synthesizable transactors. In the general case, the DUT clocks will be suspended at times to complete all the communication requests for a given point in the model execution. So at any point of time, there could be inflight data in the link.

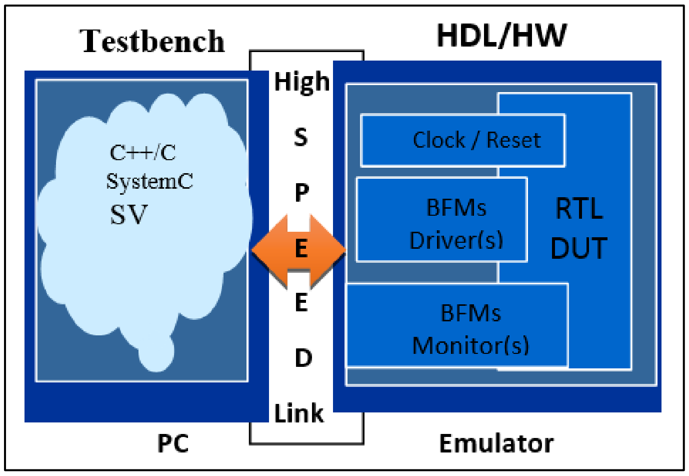


Figure 1. Co-Modeling Architecture

B. *Review of DMTCP flow*

DMTCP is an open-source package that provides a capability for Checkpoint/Restart in applications involving multiple processes/threads distributed across multiple hosts and connected by socket connections. The package can be downloaded from:

http://dmtcp.sourceforge.net

It operates under Linux, with no modifications to the Linux kernel or to the user code, and it can be used by unprivileged users (no root privilege needed). One can later restart from a checkpoint, or even migrate the processes by moving the checkpoint files to another host prior to restarting. Figure 2 shows the typical flow of a user job under DMTCP.

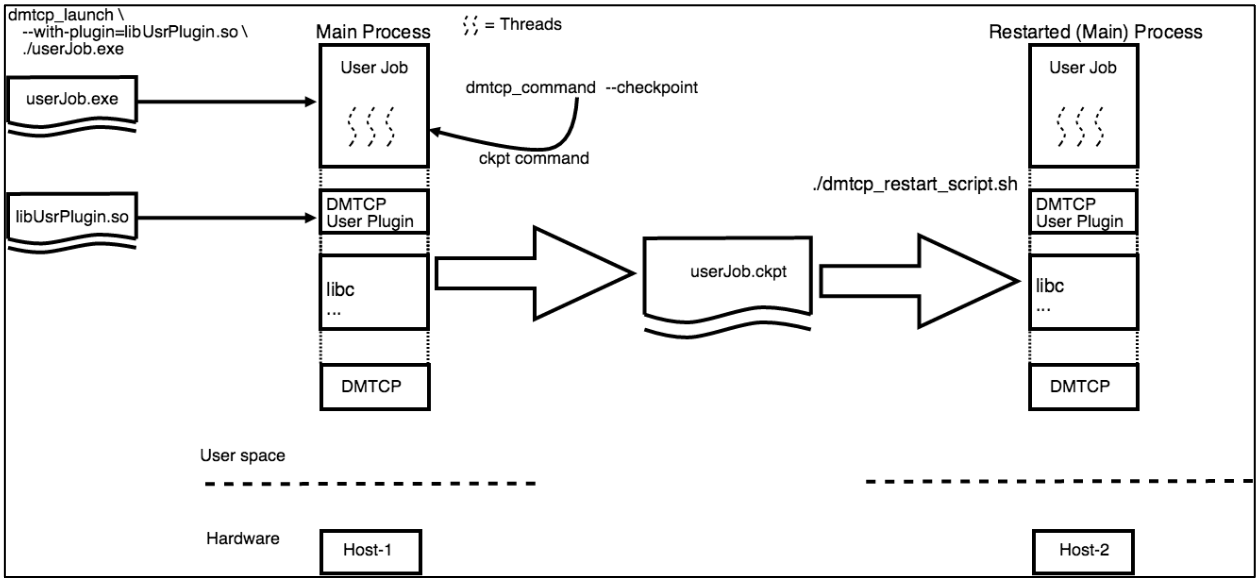


Figure 2. Typical flow of a user job under DMTCP

A detailed description on DMTCP (Distributed MultiThreaded Checkpointing) internals can be found in [1]. DMTCP also provides a flexible plugin model that supports the ability to write an add-on library that can: support DMTCP event hooks; add custom wrappers around system calls; and add a custom distributed name service facility [2, 3].

III. MOTIVATING USE CASES

Checkpoint-restore of emulation jobs opens a wide of range of applications, which will help users build a fault-tolerant emulation system, powerful debug mechanisms, and increase usage efficiency of critical resources like hardware emulators. This section describes different use cases for such capability in emulation. The following section, Section IV, then presents the Checkpoint/Restore Framework in detail.

A. *Skipping repeated initial sequences*

Designs may have an initialization phase that is always executed for each test. This may be a hardware reset phase or boot-up, which takes a great deal of time before an actual test can start. We can save much of the emulation runtime by taking a checkpoint right after this repeated initial sequence. New tests can then just restart immediately after this initial sequence, thus saving a great deal of regression time. Another application of this is to do what-if analysis after reaching an interesting point in the execution.

In Figure 3, each test *Test0*, *Test1*, *Test2* executes the same initial sequence, which takes *C1* time.

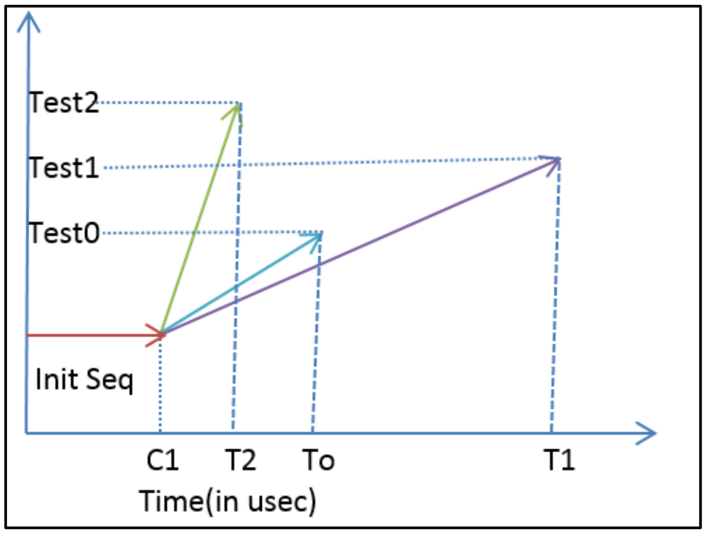


Figure 3. Job Progress with fixed initialization sequence

The tests then follow their own unique paths, completing the test in different time intervals (*T0, T1, and T2*). This *C1* time is saved if we checkpoint just after completion of the initial sequence and then restore from the checkpoint at the original state in each test.

B.   Better Job Management Policies

One of the advantages of virtualization of test environments is the ability to use emulators remotely. This allows emulation to be moved to the data center, with jobs being managed by a workload management platform such as LSF. However, the non-pre-emptive nature of emulation jobs forbids pre-emptive scheduling policies. This prevents a high-priority job from acquiring the resources occupied by a currently executing low-priority job. For example, a currently running long job cannot be removed even though a short job has just arrived. The short job has to wait until the long job exits and frees up the resources. Hence, the non-pre-emptive nature of emulation jobs can easily lead to inefficient use of emulators.

Consider the scenario depicted in Figure 4:

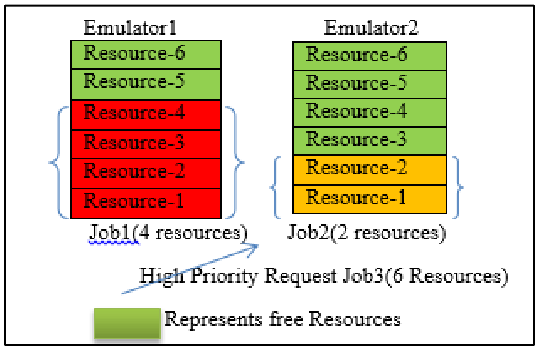


Figure 4. Job Management

Job1 occupies 4 slots on *Emulator1* and *Job2* occupies 2 slots on *Emulator2*. Further a high-priority request from *Job3* arrives for 6 resources. As resources are fragmented, irrespective of its priority, *Job3* has to wait for either of *Job1* or *Job2* to finish.

The Checkpoint/Restore capability can remove this shortcoming and enable a pre-emptive scheduling policy. We could have checkpointed either of *Job1* or *Job2* and freed up the resources for *Job3*. Whenever resources become available, the checkpointed job can then be restarted using the DMTCP restart scripts.

Further, this situation also leads to inefficient use of emulators due to resource fragmentation within each emulator. Using Checkpoint/Restore we could have migrated an interfering job to a different emulator, thereby making contiguous resources available for the new job.

In Figure 5, the checkpoint of *Job2* is taken first, and then *Job2* is restarted on *Emulator1*. This frees up all 6 slots on *Emulator2* for *Job3*. *Job3* can now be allocated using contiguous resources on *Emulator2*.

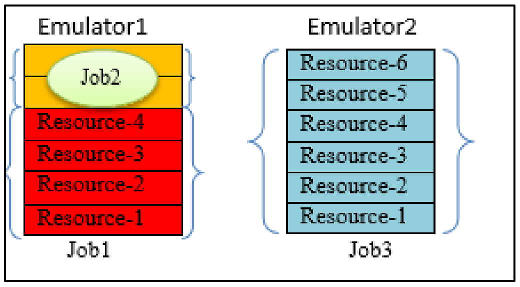


Figure 5. Resource free with Job Migration

Another interesting example concerns the issue of "fairness" for large-capacity jobs. In a distribution system dominated by high-priority, small-capacity jobs, a low-priority large-capacity job may never actually get a chance to run if the policy is to wait for required slots to become available. Further, a high-priority long job can lead to underutilization of emulation resources as freed slots cannot be allocated for other small jobs if they have to be pooled for a high-priority long job waiting in the queue. With checkpoint-restart, one can save all the small jobs at once, run the large job and then restart the small jobs. A long low-priority job might otherwise never get an opportunity to complete, if not for the use of checkpoint-restart. There are several such instances where checkpoint-restart can offer this kind of flexibility in an emulation job scheduling system.

C. *Debugging from past simulation time*

Debugging is an important requirement for verification engineers. Much time is spent in debugging functional issues in design as well as integration issues with software during validation of a full SoC. The ability to take a checkpoint of the full system can provide capabilities to start debugging just before an issue occurs, by restarting from the point of a previous checkpoint state. For example, one can take periodic checkpoints and when a problem is seen: start from the last checkpoint, run again and capture more debug information. This is a tremendous advantage when a debugging issue occurs only after a run of long duration. A large amount of time is saved, since one no longer needs to wait for the test to arrive at the point of interest.

IV. CHECKPOINT/RESTORE FRAMEWORK

In this section we describe the implementation of checkpoint-restore for emulation jobs. We address checkpoint and restore separately. Figure 6 shows the typical flow of an emulation job under DMTCP. Note that in Figure 2, the checkpoint was invoked by the user at an arbitrary point in time. However, in the emulation flow (Figure 6), the user invokes the checkpoint programmatically from within the user code, when the simulation reaches certain number of cycles.

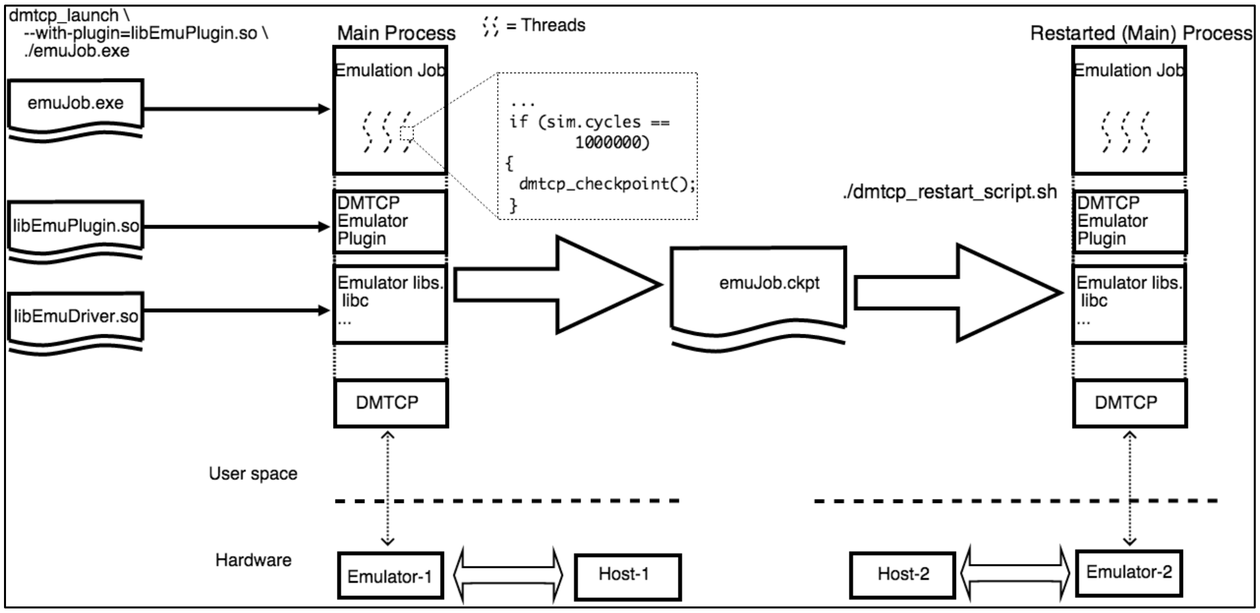

```
dmtcp_launch \
  --with-plugin=libEmuPlugin.so \
  ./emuJob.exe
```
```
...
if (sim.cycles ==
       1000000)
{
  dmtcp_checkpoint();
}
```
```
./dmtcp_restart_script.sh
```

Figure 6. Typical flow of an emulation job under DMTCP.

A. *Checkpoint*

Checkpointing of emulation jobs involves both the emulator and the testbench on the workstation. One must save the simulation state for the design running on an emulator, while also employing binary-level checkpointing using DMTCP for the processes representing the testbench side and running on the workstation. As always, we have to make sure that the hardware side stops generating further transactions when a checkpoint is in progress. In addition, checkpointing requires that we make sure that there is no in-flight data inside the high-speed interface between the emulator and the testbench at the time of checkpointing. In the case that a checkpoint is taken without flushing the in-flight data, those transactions will be lost at the time of restoring from the checkpoint, thus resulting both in incorrect hardware state and incorrect state on the testbench side.

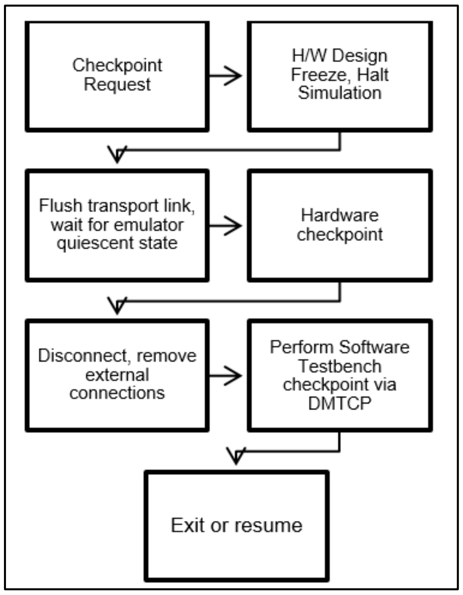

Checkpoint Request
H/W Design Freeze, Halt Simulation
Flush transport link, wait for emulator quiescent state
Hardware checkpoint
Disconnect, remove external connections
Perform Software Testbench checkpoint via DMTCP
Exit or resume

Figure 7. Flowchart for Checkpoint Algorithm

The following is the integrated algorithm for checkpointing of an emulation job. These steps are also depicted graphically in Figure 7.

1) A *Checkpoint Request* is received by the testbench either through an external utility such as *dmtcp_command* or through a user-exposed API, *ScheduleCheckpoint*, made by testbench itself. The request is sent to the emulator in order to freeze the running design. This will stop both the user design clocks and the simulation time itself.

2) Once a design is frozen, additional transactions from the hardware side will no longer be generated. At this point the data in flight inside the link is flushed until all of them have been processed and the emulator reaches a quiescent state.

3) When the emulator has reached a quiescent state, the system initiates the checkpoint of the emulator model by using the checkpoint technology native to the emulator. This will store the current design state to disk.

4) The system then disconnects from the emulator and makes sure that all emulator-related processes are terminated. This step will ensure that any external connections not running under DMTCP are removed from consideration. For example, there may be connections to a licensing server, to some waveform dump servers, and so on. This also reduces the overall checkpoint time by freeing up memory used by these external connections that the binary checkpointing procedure would otherwise have to save, as described in the next step. Further, checkpoints taken this way are independent of which specific emulator the session was running on, and this makes it easy to relocate saved emulation jobs to other emulators.

5) Next, the system *initiates* DMTCP-based checkpointing on the testbench side. This involves calling *dmtcp_checkpoint*, a part of the DMTCP API. Here, one writes an emulation-specific external plugin for DMTCP, which will specify which files must be checkpointed. The DMTCP plugin is specific to the emulator and testbench infrastructure, and must also specify the path to the checkpoint database, where DMTCP checkpoint image files are saved alongside the hardware database, as a consolidated database within the user specified path. . DMTCP also allows a user to have certain processes be explicitly excluded from checkpointing. This might be required for the case where processes have been spawned from an external library linked into the user testbench. Those processes are not required as part of a correct checkpoint state. And further, attempting to save such superfluous processes leads to

other issues when restoring from a checkpoint if those processes were communicating with external processes that were not running under DMTCP. The details of DMTCP external plugins can be found in [2, 3].

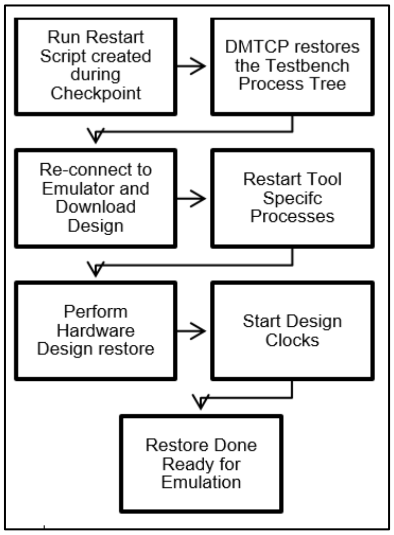
6) After the invocation of *dmtcp_checkpoint* returns, we can choose either to exit the current emulation or to resume the current run.

7) After resuming from a checkpoint, the following steps are taken:

- *Re-connect* to same emulator and configure the same design again.
- *Restart* the tool-specific processes and re-connect to servers from which they exited in step (4). This step is isolated to the emulation tool's internal workings, and does not require the collaboration of DMTCP.
- *Perform* a hardware design restore from the same checkpoint database and start the design clocks.
- Note that after being restored from a checkpoint, the testbench side resumes in a correct state at the end of this procedure, and so nothing more is required on the testbench side.

B. *Restore/Restart*

The DMTCP package dumps a restart script at the time of checkpoint. This script takes care of restarting the entire testbench tree of processes. This script also takes care of restarting processes that were on remote machines. After restart, the system will "wake up" in step (6) in the checkpoint sequence above, as if we have just returned from the call to *dmtcp_checkpoint*. So the remaining steps to restart are the same as those described in step (7) above.


Run Restart Script created during Checkpoint
DMTCP restores the Testbench Process Tree
Re-connect to Emulator and Download Design
Restart Tool Specifc Processes
Perform Hardware Design restore
Start Design Clocks
Restore Done Ready for Emulation
Figure 8. Flowchart for Restore

The following is the algorithm for the restart/resume. These steps are also depicted graphically in Figure 8.

1) Use the restart script generated by DMTCP that relies on the checkpoint database (the checkpoint image files) in order to bring the whole testbench tree of processes up and running.

2) After DMTCP's restore/restart, we return from the call to *dmtcp_checkpoint*. The next steps bring the HDL side up and fork any tool-specific processes.

3) The system then re-connects the emulator resource and downloads the design.

4) The system then restarts the tool-specific processes and re-connects to external servers such as license servers, waveform collection servers, etc. This step is completely isolated from the internal workings of the emulator tool.

5) The system then performs the hardware design restore using the same checkpoint database, and it starts the design clocks. This is the last step in successfully restoring the emulation.

V. Case Study: Skipping the OS boot in an OEM company's SoC validation environment

For this case study, the SoC validation environment instantiates a SoC containing a CPU, Memory Subsystem, Switching Fabric, and peripherals. The SoC is modeled in a hybrid environment whereby part of the SoC is modeled on the workstation and part of the SoC is modeled in the emulator, and the Switching Fabric is the bridge between the models.

Many of the SoC validation use cases require firstly booting an operating system (OS) in order to run the applications that are being validated. The boot of the OS takes on the order of hours to days. Often the applications that are being validated execute in a fraction of the time that it takes to boot the OS. This means that only a fraction of the emulation time was used to validate the application and the larger portion of the time is spent booting the OS in order to be able to run the application. Thus the overall boot time has an important impact not only on how many applications can be run per user per day, but also, from an emulation efficiency standpoint, how much emulation time is spent just getting the OS booted versus running the applications that are used to validate the SoC and/or the application.

The ideal scenario would be to eliminate the time it takes to boot the OS. A solution close to the ideal scenario is to checkpoint the SoC validation environment after the boot of the OS. This OS boot can then be delivered as part of a checkpoint image that is bundled with the rest of the SoC validation environment. The checkpoint of the hardware can be taken by technology native to the emulator. The complexity is in the checkpointing of the part of the SoC validation environment that is executing on the workstation.

There were two early attempts to checkpoint, before settling on DMTCP as the preferred solution. A first attempt at checkpointing, prior to the use of DMTCP, was to capture all of the stimulus from the hardware during the OS boot along with a checkpoint of the hardware, and then to restore the OS boot using the stimulus. This stimulus was replayed into the software test bench to re-establish the state of the software, and then finally restore the state of the emulator using the hardware checkpoint.

This method worked, except it had two notable drawbacks:

(1) Depending on how much stimulus needed to be captured, the size of the replay database could become quite large.

(2) The time it took the software testbench to execute, controlled the time it took to perform the restoration of the software testbench.

A second attempt at checkpointing prior to employing DMTCP was to leverage the Boost C++ libraries to make each of the software components of the SoC validation environment checkpoint-able. This method worked except that it had a major drawback in that each SoC validation software component had to be developed with checkpointing in mind and if there was just one component that didn¹t support checkpointing or did the checkpointing incorrectly, the SoC validation environment was not checkpoint-able. Hence, the ideal solution would be to transparently checkpoint the software in the same way as we checkpoint the hardware.

The DMTCP-based approach was able to overcome the limitations of these first two attempts by transparently checkpointing the software on the workstation, which includes the part of the SoC modeled on the workstation, the software testbench, and the emulation software. The checkpoint is taken without concern for how the software has been modeled and also without concern for the speed at which the software testbench executes. This resulted in making an "OS boot" checkpoint available to the users along with the SoC verification environment. This allows those users to restore the checkpoint in less than 5 minutes, and to then to run their applications for SoC validation in an environment after the OS boot. Now, users requiring this use case can focus their emulation time on running their application, and skipping the lengthy OS boot time. The DMTCP-based approach has accelerated the time to run an application by close to a factor of two. Additionally, it has freed up the emulation time that would have been spent in "OS boot", thus saving many hours of emulation time.

In addition, the DMTCP approach also saves the state of the emulation runtime software itself. This feature provides added value, as compared to the two earlier checkpointing approaches. The checkpoint-restore flow can now set up to address additional aspects of the emulation runtime environment, such as triggers, which are important for debugging in the case of an exception. This enables the designer to create an environment closer to a "turnkey solution", in which the end user no longer has to remember to load the trigger prior to starting the run of their application.

First use case of DMTCP at an OEM company was on a regression suite consisting of 40 jobs. The 40 jobs had a similar profile in that they required around 1 hour to boot the OS and then 1 hour of execution of test content. DMTCP was used to create a checkpoint just after the boot of the OS. This checkpoint then becomes part of the collateral of that database. Any future user of that database and software configuration can restore the checkpoint rather than re-running the boot of the OS. The restoration of database using the checkpoint takes around 5 minutes. That is a time savings of 55 minutes per job for this regression. As a reminder, this environment is a hybrid environment and pending the configuration of whether the CPU is in software or RTL and also the type of OS can greatly impact the "OS boot" time from an hour to days. For this regression the 40 jobs were able to run in around 22 hours versus previously they would have taken 40 hour. Job throughput was increased close to 2x. This results not only in a substantial time savings but also a substantial cost savings.

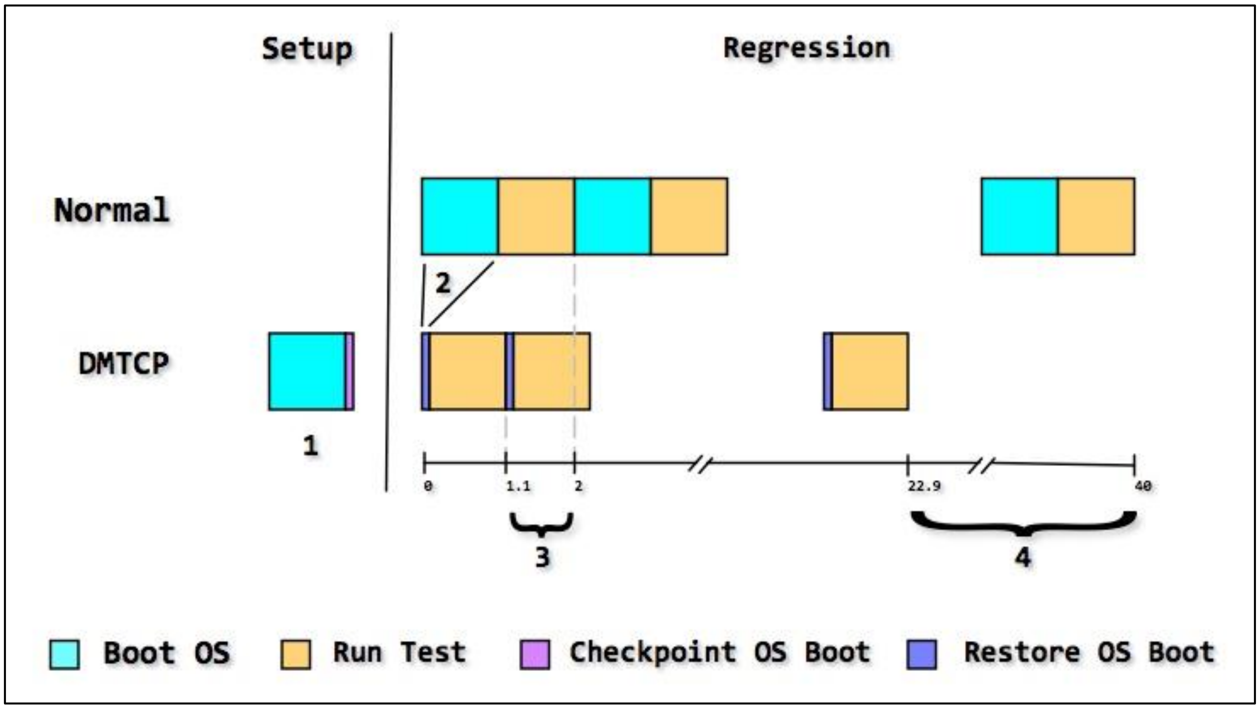


Figure 9. Regression Suite Emulation Time Comparison

Figure 9 is highlighting the changes introduced by adding DMTCP into this use case; the following points are related to the numbers in the figure 9.

1) Checkpoint is created for OS boot as part of preparing the database for users.
2) Restoring OS boot from checkpoint saves 55 minutes for each job.
3) First job is completed after 1.1 hours with DMTCP versus 2 hours without DMTCP.
4) Full regression is completed after 22.9 hours with DMTCP versus 40 hours without DMTCP.

During the integration of DMTCP with the emulation and SoC validation environment, we faced several challenges:

(1) Very large read-only files were being saved as part of the checkpoint. This had an impact on both checkpoint/restore time as well as the memory footprint on disk. This was solved by having the DMTCP Emulation plugin detect and decide not to checkpoint such read-only files. The tool-specific file list was written into the DMTCP plugin to identify just those files that needed to be checkpointed.

(2) Checkpoints were not portable to another site, due to some file paths that were preserved as part of the checkpoint. This was solved by using the existing file path virtualization plugin of DMTCP, allowing these paths to be changed at restore time.

Limitations found when deploying DMTCP

1) Some use cases have a remote process connected via TCP on Windows machine. Additional work would be required to support this kind of use case both within the DMTCP community as well as in the testbench infrastructure.

2) Some use cases require changing the initial state of the environment which is part of the DMTCP checkpoint. For instance, if you had a SoC that had a programmable number of DIMMs and the number of DIMMs was changing per test you wouldn't be able to re-use a single checkpoint.

DMTCP was found to be easy to integrate, and it required minimal changes to emulation environment. This has demonstrated the success of this approach toward OS-based use case validation during emulation.

VI. FUTURE WORK

We intend to work on developing preemptive capabilities for emulation jobs in a workload management system such as LSF. A case study in this environment is needed in order to discover the practical challenges thereof, and to help deploy this technology for more flexible job scheduling management.

Finally, verification of a network switch presents an additional interesting case study for the future. In this scenario, a virtual machine is often required, so that simulated Ethernet traffic can be injected by the virtual machine into the environment for verification test coverage. This makes transparent checkpointing on the testbench side more difficult, since even though the virtual machine can be modified to inject Ethernet traffic, the virtual machine snapshot facility is difficult to modify, and so the state of the Ethernet traffic generator will not be checkpointed. On restore, some steps in the state diagram for Ethernet packets may be lost. As part of future work, it is proposed to use the ability of DMTCP to checkpoint a virtual machine *from the outside*.

DMTCP has the ability to carry out a snapshot from the outside for the case of a *QEMU* virtual machine over KVM [4]. That same work [4] also presents DMTCP's ability to checkpoint a network of virtual machines. This latter case makes possible verification for end-to-end Ethernet test coverage between two virtual machines.

VII. CONCLUSION

This paper describes the approach to transparently checkpoint/restore emulation jobs and various key benefits it brings along with it. This integration was successfully tried in an OEM company’s SOC validation environment, which not only reduced total regression time but also increased Emulator efficiency. Emulation time is precious and any saving of this time directly affects one’s total verification cost.

Experimental results have shown that this approach, when integrated with job management system, has increased the emulator utilization and has also increased the productivity of the verification engineers by providing them with a window to look back in time. The integration has also been successfully tried for a variety of software testbenches including C/C++/SystemC, and a SystemVerilog testbench running on a simulator. These testbenches can have multiple threads or multiple processes spread across different machines.

REFERENCES

[1] J. Ansel, G. Cooperman, and K. Arya. “DMTCP: Scalable user-level transparent checkpointing for cluster computations and the desktop”. In Proc. of IEEE International Parallel and Distributed Processing Symposium (IPDPS-09, systems track). IEEE Press, 2009. Published on CD; version also available at http://arxiv.org/abs/cs.DC/0701037; software available at http://dmtcp.sourceforge.net.

[2] DMTCP team. Tutorial for DMTCP plugins, accessed Dec.10, 2013. http://sourceforge.net/p/dmtcp/code/HEAD/tree/trunk/doc/plugin-tutorial.pdf

[3] Kapil Arya, Rohan Garg, Artem Y. Polyakov and Gene Cooperman, “Design and Implementation for Checkpointing of Distributed Resources using Process-level Virtualization”, Proc. of IEEE Int. Conf. on Cluster Computing (Cluster'16), pp. 402--412, Taipei, Taiwan, IEEE Press, Sept., 2016.

[4] Rohan Garg, Komal Sodha, Zhengping Jin and Gene Cooperman, “Checkpoint-Restart for a Network of Virtual Machines”, Proc. of 2013 IEEE Computer Society International Conference on Cluster Computing, 8 pages, Indianapolis, USA. Sept., 2013.