ZeBu® Checkpoint and Restart Application Note

Version V-2024.03-1, July 2024

Feedback

Contents

Preface

This section contains the following topics:

- About This Book
- Contents of This Book
- Related Documentation
- Synopsys Statement on Inclusivity and Diversity

About This Book

The *ZeBu® Checkpoint and Restart - Application Note* provides details of various DMTCP features.

Contents of This Book

The *ZeBu® Checkpoint and Restart - Application Note* has the following chapters:

Chapter	Describes...
Introduction to Checkpoint	Description of DMTCP
Installing the DMTCP Tool	Installation and Validation of DMTCP Tool
Performing a Checkpoint	Description of various methods to perform checkpoint
Checkpoint Directory	Details of checkpoint directory
Restarting Emulation from a Checkpoint	Procedure to restart emulation
Converting fastState to hardwareState Between DMTCP Save and Restart	Converting fastState to hardwareState Between DMTCP Save and Restart
Debug on Restart	Describes debug DMTCP during restart
Limitations	Limitations of DMTCP
7 ZIP Support	7zip support for checkpointing

Feedback

Chapter	Describes...
Forked Checkpointing	Checkpointing of SW part in a background process
Checkpointing and Restore for Transactors	Checkpointing for transactors
Use Models	Use models examples

Related Documentation

Document Name	Description
ZeBu User Guide	Provides detailed information on using ZeBu.
ZeBu Debug Guide	Provides information on tools you can use for debugging.
ZeBu Debug Methodology Guide	Provides debug methodologies that you can use for debugging.
ZeBu Unified Command-Line User Guide	Provides the usage of Unified Command-Line Interface (UCLI) for debugging your design.
ZeBu UTF Reference Guide	Describes Unified Tcl Format (UTF) commands used with ZeBu.
ZeBu Power Aware Verification User Guide	Describes how to use Power Aware verification in ZeBu environment, from the source files to runtime.
ZeBu Functional Coverage User Guide	Describes collecting functional coverage in emulation.
Simulation Acceleration User Guide	Provides information on how to use Simulation Acceleration to enable cosimulating SystemVerilog testbenches with the DUT
ZeBu Verdi Integration Guide	Provides Verdi features that you can use with ZeBu. This document is available in the Verdi documentation set.
ZeBu Runtime Performance Analysis With zTune User Guide	Provides information about runtime emulation performance analysis with zTune.
ZeBu Custom DPI Based Transactors User Guide	Describes ZEMI-3 that enables writing transactors for functional testing of a design.
ZeBu LCA Features Guide	Provides a list of Limited Customer Availability (LCA) features available with ZeBu.
ZeBu Synthesis Verification User Guide	Provides a description of zFmCheck.

Document Name	Description
ZeBu Transactors Compilation Application Note	Provides detailed steps to instantiate and compile a ZeBu transactor.
ZeBu zManualPartitioner Application Note	Describes the zManualPartitioner feature for ZeBu. It is a graphical interface to manually partition a design.
ZeBu Hybrid Emulation Application Note	Provides an overview of the hybrid emulation solution and its components.

Synopsys Statement on Inclusivity and Diversity

Synopsys is committed to creating an inclusive environment where every employee, customer, and partner feels welcomed. We are reviewing and removing exclusionary language from our products and supporting customer-facing collateral. Our effort also includes internal initiatives to remove biased language from our engineering and working environment, including terms that are embedded in our software and IPs. At the same time, we are working to ensure that our web content and software applications are usable to people of varying abilities. You may still find examples of non-inclusive language in our software or documentation as our IPs implement industry-standard specifications that are currently under review to remove exclusionary language.

Feedback

1

Introduction to Checkpoint

The checkpointing feature allows you to save the status of the entire hardware DUT and the software testbench environment during the emulation runtime. The most common scenario is to avoid using valuable emulation cycles to get the previous system boot/initialization when running software tests.

When hardware state is natively captured by ZeBu Save and Restore functionality, saving the software testbench environment is done using an external open-source tool called Distributed MultiThreaded Checkpointing (DMTCP).

This section discusses the following topics:

- Customized DMTCP Tool

Customized DMTCP Tool

The DMTCP tool is introduced in ZeBu to save an emulation hardware and software state to restart the emulation later. This tool enables you to restart the emulation before a point of interest (for example, after an OS boot sequence) for debug purpose.

The ZeBu release includes a customized version of DMTCP at `$ZEBU_ROOT/thirdparty/dmtcp/`. You should always use the DMTCP package provided with ZeBu Software release at `$ZEBU_ROOT/thirdparty/dmtcp/`. If a newer DMTCP package is acquired from Synopsys, replace the default one at `$ZEBU_ROOT` before the installation.

DMTCP OS/kernel and gcc compatibility requirements must be aligned with ZeBu Software installation requirements, which are described in Qualified System Configuration (QSC) for every ZeBu release.

If the DMTCP is used with a test application, which utilizes both 64 bits and 32 bits libraries, the DMTCP should be recompiled with the `-m` option. This enables the support for such multi-arch applications.

2

Installing the DMTCP Tool

The DMTCP tool must be built locally on the same host that is used for DMTCP runtime. DMTCP OS/kernel requirements are aligned with ZeBu Software QSC installation requirements.

This section discusses the following topics:

- Installation Instructions
- Additional Variables for DMTCP
- Validation of DMTCP Installation

Installation Instructions

You must follow the instructions included in the package to install this version of DMTCP. If the `ZEBU_ROOT` is set, the DMTCP tool can be installed by default using the following command:

```
./install_dmtcp.sh -d
```

By default, DMTCP is installed from the `${ZEBU_ROOT}/thirdparty/dmtcp` directory.

Alternatively, DMTCP can be installed from a specified location using the following command:

```
./install_dmtcp.sh -d -i <path to DMTCP package>
```

Note:

DMTCP is installed at `/tmp/dmtcp_${hname}_${USER}_${abi}` and ABI is automatically detected.

Prerequisites

Before installing DMTCP, it is recommended to source ZeBu environment settings as follows:

```
source <ZEBU_INSTALLATION_DIRECTORY>/zebu_env.sh(bash)
```

DMTCP Installation Script

A script (`install_dmtcp.sh`) is available at `$ZEBU_ROOT/thirdparty/dmtcp/scripts` to automate the DMTCP installation process.

The script consists of the following options to help with installation:

- `-i <path to DMTCP install package>`: Path to DMTCP install package.
- `-m`: Switch to enable multi architecture build.
- `-p <DMTCP root installation directory>`: DMTCP root directory for installation.

  Default value: `/tmp`
- `-o <DMTCP unpacking directory name>`: DMTCP installation directory. If absolute path is given, the `-p` option is omitted.

  Default value: `dmtcp_${hname}_${USER}_${abi}`
- `-f`: Forces to override existing output directory
- `-d`: Installs DMTCP package: from `${ZEBU_ROOT}/thirdparty/dmtcp directory`, in `/tmp/ dmtcp_${hname}_${USER}_${abi}` directory, detects correct `ABI` for DMTCP compilation. `ABI` version can be overwritten by the `-n` or `-l` option.
- `-n`: Sets `ABI` to `cxx11`.
- `-l`: Sets `ABI` to `cxx03`.
- `-t <threads>`: Sets the number of threads used for the DMTCP compilation.

  Default value: `Max thread`
- `-b`: Does not modify `PATH` to point to a Linux system `gcc`.
- `-r`: Runs DMTCP tests after successful installation (it is equivalent to executing "make check" during the DMTCP installation using the original DMTCP scripts)
- `-q <path to directory for log>`: Log file directory destination (Directory must exist) and it is used for `-e` and `-a` options.
- `-e <error log file name or path>`: Error log file name or path. It is only created in case of an error during installation. It can be used with `-q` option or absolute path can be used.
- `-a <log file name or path>`: This log file is the output of the script, and it is always created. It can be used with `-q` option or absolute path can be used
- `-s`: A silent mode. It prints nothing other than an error message or `DMTCP_PATH`.
- `-z`: Disables printing an error message in the silent mode.

- `-h`: Displays help

  The script provides the following advanced options to help with installation:

  - `-w`: Adds the `--enable-debug` option to DMTCP configuration.
  - `-y`: Adds the `--enable_logging` option to DMTCP configuration
  - `-u <directory path>`: Adds the DMTCP log files directory.

    Note:

    `DMTCP_TMPDIR` is set in `dmtcp_settings`.

To provide necessary additional compilation options to DMTCP, use the `DMTCP_CONFIG_EXT` variable as follows:

```
DMTCP_CONFIG_EXT="--enable-debug --enable-logging " ./install_dmtcp.sh -d
 -f
```

This script (`dmtcp_install.sh`) verifies the following before installing DMTCP:

- if DMTCP is already installed with an appropriate version, which is compatible with ZeBu.
- if new `CXX11 ABI` is used in ZeBu to configure and install DMTCP accordingly.
- if the selected `ABI` is supported with gcc used for compilation

After completing this verification, the `dmtcp_install.sh` script compiles and installs DMTCP on a workstation with Linux OS, in a local directory for a user. Subsequently, it creates a script (`dmtcp_settings.sh(csh)`) to source and the script sets essential environment variables. The script is created after successful compilation process.

Post-Installation Tasks

After successfully installing DMTCP, the following files are generated in the DMTCP output directory:

- `dmtcp_settings.sh`
- `dmtcp_settings.csh`

You must source one of the scripts based on the Linux shell before running a testcase with DMTCP. The script sets or updates the following environment variables:

- `DMTCP_HOME`: Sets to the directory where DMTCP was installed.
- `DMTCP_PATH`: Sets to the directory where DMTCP was installed.
- `PATH`: Sets this variable to a path to executable tools from DMTCP.

If `ZEBU_ROOT` is set, then this also sets `$ZEBU_ROOT/thirdparty/dmtcp/7Zip/7zzs` to the `PATH` variable.

- `LD_LIBRARY_PATH`: Sets this variable to a path to directory with DMTCP libraries.

  Note:

  Any other environment variables required by the design should be added to `custom set_env.sh` script, which is created or maintained by you.

- `ZEBU_INTERNAL_SIGNAL=PWR`: Mandatory for zRci flow since there is a collision when same Linux signal is used both by DMTCP and zRci.

Additional Variables for DMTCP

It is recommended to set additional environment variables (`envvars`) for correct usage of DMTCP tool with ZeBu Software for specific use cases as follows:

- For Hybrid/Virtualizer use case where board can be opened and closed by different threads that can lead to crashes/freezes on emulation closure, set the following mandatory variable:

  ```
  setenv ZEBU_DONT_CANCEL_THREADS true
  ```

- When **zRci** is driving the emulation (the same Linux signal is used by both DMTCP and **zRci** leading to collision), use the following mandatory variable for the **zRci** flow:

  ```
  setenv ZEBU_INTERNAL_SIGNAL PWR
  ```

- When standard memory and thread management functions are actively used by user testbench, additional ZeBu plugins are required to avoid deadlocks accessing `libc` functions (thread and memory management and so on). In this scenario, use the following:

  ```
  setenv DMTCP_PLUGIN
   $ZEBU_ROOT/lib/libpthread_join.so:$ZEBU_ROOT/lib/libthreadmalloc.so
  ```

  This environment variable is already set by default by ZeBu environment. Therefore, no additional user action is needed.

- For complex testcases, when DMTCP image is large and takes a lot of time to finish, use the following to enable DMTCP checkpoint progress bar printing:

  ```
  setenv DMTCP_SHOW_PROGRESS_BARS 1
  ```

- When there are multiple processes running under DMTCP, such as parent process runs under DMTCP and starts a new one, use the following:

  ```
  setenv ZEBU_DMTCP_ENABLE_MULTI_PROCESS 1
  ```

Validation of DMTCP Installation

After installation, it is important to validate correctness of DMTCP tool using the simple example represented as follows:

1. Create a simple C++ source file as `dmtcp_test.cc`:

```
#include <iostream>
using namespace std;

int main(int argc, char **argv)
{
        int exit;
        cout << "Before checkpoint" << endl;

        while (true) {
/* Entry point for execution termination (CTRL+C) after checkpointing:
 restart will proceed from here */
                cin >> exit;

                cout << "After checkpoint" << endl;
                if ( exit == 0 ) break;
        }

        cout << "End of test" << endl;

        return 0;
}
```

2. Compile the executable as follows:

```
g++ dmtcp_test.cc -o dmtcp_test
```

3. Launch `dmtcp_coordinator` in a separate shell (control shell):

```
dmtcp_coordinator
```

4. Launch `dmtcp_test` executable using `dmtcp_launch` in the second shell (launch shell):

```
dmtcp_launch ./dmtcp_test
```

"**Before checkpoint**" message should be displayed in the launch shell.

1. Use the control shell to create a checkpoint: type `'c'` and confirm with **ENTER**.
2. Type `'q'` in the control shell to terminate both `dmtcp_launch` and `dmtcp_coordinator`.

You should see `dmtcp_restart_script.sh` created in the working directory, and the `*.dmtcp` file with checkpoint data.

1. Launch `dmtcp_restart_script.sh` using the launch shell to restart the test application:

   ```
   ./dmtcp_restart_script.sh
   ```

2. Type `'0'` and confirm with **ENTER**.

   The "**After checkpoint**" and "**End of test**" messages are displayed to confirm the successful test program execution.

3. Restart script can be used several times to continue test application execution from the same checkpoint created.

Feedback

3

Performing a Checkpoint

This section discusses the following topics:

- Performing a Checkpoint Using zRci Flow
- Performing a Checkpoint Using C++ Method

Performing a Checkpoint Using zRci Flow

Prerequisites

All form of traffic should be stopped (DPI, FWC, QiWC) before doing a checkpoint with **zRci**. To disable each feature, use the following commands:

```
ccall -disable
dump -disable
dump -close
run -disable all
checkpoint -fullsave <checkpointdir>
```

Procedure

To checkpoint with the **zRci** flow, perform the following steps:

1. Run **zRci** using `dmtcp_launch`:

   ```
   dmtcp_launch zRci --do <run.tcl>
   ```

2. At any point during emulation after `start_zebu`, use the following command to create a checkpoint (full hardware and software state):

   ```
   checkpoint -fullsave
   ```

3. This command gives the following return value:

   - `DMTCP_RESUME` (or 1) when the command is from the session that performed the DMTCP save.
   - `DMTCP_AFTER_RESTART` (or 2) when the command is in the restored session after the DMTCP save.

For more details on checkpointing usage for **zRci** flow, see **ZeBu Server Unified Command-Line User Guide**.

Note:

Special treatment is required for user designs with transactors. This command cannot be used directly for design with transactors without proper handling of pre- and post-checkpoint callbacks. For details, see Recommendations for Multi-transactor Designs.

Performing a Checkpoint Using C++ Method

This method belongs to the class `ZEBU::Board` and it can be called without any arguments. In this case, the name of checkpoint is auto-generated. Pure C call is also available and has similar signature.

Note:

checkpoint() must be called after `Board::init()`.

```
bool checkpoint(const char *checkpointDirectory = "./checkpoint",
int *checkpointStatus = 0,
            unsigned int (*callBack_beforeCkpt) (void *context) = 0,
           unsigned int (*callBack_afterCkpt) (void *context, int
 checkpointStatus) = 0,
               void *context = 0,
               bool suffix = true) throw(std::exception);
```

where,

- `checkpointDirectory`: Spec fies the path to the directory that stores all the files required by the DMTCP feature. Each checkpoint directory created is suffixed with an integer.
- `checkpointStatus`: Specifies the checkpointing status updated by the checkpoint function. It allows you to see the **Resume** status and **Restart** status from the checkpoint. The available values are:
  - `1`: **Resume** status
  - `2`: **Restart** status
- `callBack_beforeCkpt`: Specifies the callback function that is executed before the actual checkpoint.

- `callBack_afterCkpt`: Specifies the function that provides the context call after a checkpoint. This function is executed twice:
  - Once during the resume
  - Once during the restart

  Note:

  The `checkpointStatus` argument of this callback is the same as the checkpointing status mentioned for the checkpoint method.
- Context is a pointer to an argument that is passed to both `callBack_beforeCkpt` and `callBack_afterCkpt` callbacks.

  Note:

  You must evaluate this callback function to handle `prepareCheckPoint()` and `resumeCheckpoint()` transactor APIs accordingly if there are any transactors from Synopsys used in the design. This also applies to user custom-made ZEMI3 transactors. You must evaluate those callback functions to make sure all communication between testbench and hardware is completed before doing a checkpoint.
- `Suffix` is a flag used to expand the desired checkpointing directory name with an integer when it is set to `true`. It is enabled by default.

Performing a Checkpoint Within a Testbench

Perform the following steps for checkpointing within a testbench:

1. Run the testbench with the DMTCP tool using the following command:

   ```
   dmtcp_launch --modify-env testbench <arguments>
   ```

This starts a DMTCP daemon that handles the DMTCP requests.

If you do not intend to change the environment variables (see Managing Environment Variables at Restart) at restart time. You can skip the `--modify-env` option.

When using DMTCP on a testbench that uses Hardware Trigger, it is recommended to set the following:

```
setenv CXX "dmtcp_nocheckpoint g++"
```

1. Call the C++ method within the testbench, as described in Performing a Checkpoint Using C++ Method.

This call creates a checkpoint directory that is the first parameter of the checkpoint method.

Feedback

4

Checkpoint Directory

This section discusses the following topics:

- Description of Checkpoint Directory
- Renaming a Checkpoint Directory
- Naming Format for DMTCP Directories

Description of Checkpoint Directory

The generated checkpoint directory is suffixed with an integer that increments at every checkpoint without modifying the directory name.

For example:

```
save_directory.0
save_directory.1
save_directory.2
```

When a checkpoint is performed, each generated directory contains the following items:

- `fastState`: Specifies the directory containing the data related to the status of the design.
- `dmtcp_testbench_name_xxx.dmtcp`: Specifies the file containing the data related to the software processes.
- `dmtcp_restart_script_xxx.sh`: Specifies the script generated by DMTCP to reload the DUT without any change in the environment variables.
- `dmtcp_restart_script_zebu.sh`: Specifies the script used to reload the software context and restart the emulation runtime while taking into account the environment variables. For more information on these environment variables, see Managing Environment Variables at Restart.

After the checkpoint is performed, the emulation runtime resumes.

Renaming a Checkpoint Directory

Before doing a restart, the generated checkpoint directory can be renamed or moved within the directory used to run emulation. The initial checkpoint must be done with plugin that allows environment modifications (`--modify-env` parameter passed to `dmtcp_launch`). To rename the checkpointing directory, perform the following steps:

1. Rename or move the checkpoint directory.
2. Change all the occurrences of the old name in the scripts within the checkpoint directory.
3. Restart (from the previous run directory) calling `dmtcp_restart_script_zebu.sh` from new checkpoint directory as follows:

   ```
   ./<new_ckpt_dir>/dmtcp_restart_script_zebu.sh
   DMTCP_RESTART_DIR=<new_ckpt_dir>
   ```

   where `DMTCP_RESTART_DIR` contains path to a new checkpoint directory.

   Note:

   The original run directory must be used. Renaming checkpoint directory feature allows only to rename checkpoint directory, not the runtime one.

   Manual modification in `dmtcp_restart_script_zebu.sh` and `dmtcp_restart_script_xxx.sh` scripts must be done to change the old name to the new one.

   When restarting from checkpoint that was created from the renamed checkpoint, the `DMTCP_RESTART_DIR` must be empty if you do not intend to rename the folder.

Naming Format for DMTCP Directories

`Board::SetCheckpointNameType()` allows to choose the naming format for the DMTCP checkpoint directories as follows:

```
bool Board::SetCheckpointNameType(int type);
```

Where,

- When `type` is 0, checkpointing creates directories using the legacy CVS style naming (suffixed with an integer).
- When `type` is 1, checkpointing creates directories using the exact name provided as the first parameter of the *Board::checkpoint()* function.

This function returns true on success. By default, `type` is set to 0.

Note:

Multiple calls to this function with values associated with multiple calls to `Board::checkpoint()` result in unpredictable naming of checkpoint directories.

Reducing the Footprint of Checkpoint Directory

You can reduce the footprint of checkpoint directory in several ways when disk space is a crucial factor in DMTCP checkpointing usemodel. When the software image footprint (`.dmtcp`) is reduced using the 7-zip compression engine, the hardware image size can be reduced using one of the following approaches:

- Auto-removal of SLR readback files from the `fastState` directory
- 7zcompress.sh/7zdecompress.sh for `fastState` directory
- xor_checkpoint.sh for `hardwareState` directory

Auto-removal of SLR Readback Files from the fastState Directory

During conversion from fast hardware state (`fastState`) to hardware state (`hardwareState`), the SLR readback files from `fastState` (`fpga_Ux_*`) are no longer needed. These files can be auto-cleaned during the conversion by setting the `ZEBU_DMTCP_FASTSTATE_SLR_REMOVAL` environment variable.

7zcompress.sh/7zdecompress.sh for fastState Directory

When checkpoint is not intended for immediate restart, the `fastState` directory can be temporarily compressed using the 7-zip utility for saving disk space. The directory can be decompressed before restart to retrieve the original `fastState` files. The compression and decompression can be done using the `7zcompress.sh`/`7zdecompress.sh` scripts available at: `$ZEBU_ROOT/thirdparty/dmtcp/scripts/`.

Perform the following steps to compress/decompress the directory:

1. Take a backup of the checkpoint directory.
2. Apply 7-zip `fastState` compression/decompression.
   a. Make sure 7-zip binary is available in the `PATH` environment variable.
   b. Apply compression: `7zcompress.sh <path_to_fastState_dir> ./new_fast_state`

      After compression, the following log information is available:

      ```
      Creating archive: ./new_fast_state.tar.7z

      Add new data to archive: X file
      ```

```
Files read from disk: X
Archive size: …bytes (… MiB)
Everything is Ok
Size of original directory: … …M      ./fastState/
Size of compressed file   : … …M        ./new_fast_state.tar.7z
Compression Ratio: …
Compression Percentage: …%
Time taken to compress    :
                 real time: …s
                 user time: …s
                 sys time : …s


- Decompression:
 7zdecompress.sh ./new_fast_state.tar.7z ./fastState_1

After the compression you should see similar log information:
7zdecompress.sh version:1.0
… list of files …
real    …s
user    …s
sys     …s
Size of decompressed directory: …
Size of compressed file   : …
Compression Ratio: …
Compression Percentage: …%
```

3. Rename the `fastState` directory to `fastState_orig`, then rename the `fastState_1` directory to `fastState`.
4. Remove the `hardwareState` directory.
5. Run restart to verify the changes.

xor_checkpoint.sh for hardwareState Directory

To reduce hardware image footprint when checkpoint is not intended for immediate restart, bitfiles from the `hardwareState` directory can be temporarily replaced by their diff'ed versions. This can be done using the XOR function on the original bitfiles from the `zebu.work` compile database. Before restart, backward conversion is required to restore originals from diff'ed files. XOR conversion is done using the `xor_checkpoint.sh` script.

The script can be found at `$ZEBU_ROOT/thirdparty/dmtcp/scripts/xor_checkpoint.sh`.

Use `xor_checkpoint.sh -h` to print a brief help information.

The typical application is as follows:

- After save:
  - Take a backup of the checkpoint directory.
  - Run the following script to XOR bitfiles:

    ```
    ./xor_checkpoint.sh -c <path_to_hardwareState> -z
     <path_to_zebu_work> -x
    ```
- Before restart:
  - Run the following script to unXOR bitfiles:

    ```
    ./xor_checkpoint.sh -c <path_to_hardwareState> -z
     <path_to_zebu_work> -u
    ```
  - Run restart to verify the changes.

Note:

In both scenarios, the following log information is available:

```
NN bit files xored in XX seconds.
[INFO] hardwareState size before xor: <SIZE1> after xor: <SIZE2>
```

Feedback

5

Restarting Emulation from a Checkpoint

This section discusses the following topics:

Restarting the Emulation

To restart the emulation from a specific checkpoint, you must use the script generated in the checkpoint directory, as follows:

- **zRci** flow: `./<db_dir>/save_directory/dmtcp_restart_script_zebu.sh`
- C++ flow: `./save_directory.int/dmtcp_restart_script_zebu.sh`

Warning:

Do not perform a cd command into the `save_directory.int` directory. This causes errors when recovering data from the hardware save directory.

You must run the restart script from the same directory as the one in which you executed the original emulation.

Managing Environment Variables at Restart

The checkpointing feature in ZeBu allows you to change environment variables between the initial run and its restart. This is extremely useful to change testbench behavior after restart. Different parts of code can be scheduled for execution by changing `envvar` and using the `getenv()` call from user code accordingly.

The following behaviors are available to manage environment variables:

- By default, only the environment variables starting with `ZEBU_` (along with their value) are taken into account by the `dmtcp_restart_script_zebu.sh` script.
- To overwrite an environment variable that doesn't start with `ZEBU_` prefix on restart, set the variable to `new_value` on the same line with the restart script,

  ```
  ./dmtcp_restart_script_zebu.sh ENVVAR=new_value
  ```

  Note: The overwrite works only if the original save was done using `dmtcp_launch --modify-env <params>`.

Saving and Restoring an Emulation State

You can use the following script commands based on the interface you chose to run the job and save the emulation state:

- **zRci**: `checkpoint -fullsave <checkpointname>`
- **C++**: `Board::checkpoint ()`

To restart a checkpoint, use the following command:

```
<checkpointname>.n/dmtcp_restart_script_zebu.sh
```

Where, `n` is a placeholder for an integer.

6

Checkpointing and Restore for Transactors

When the call to `board::checkpoint()` is done in the testbench, transactor hardware and software part is also saved, which can be restored later.

However, the following transactors, which have GUI support, are handled differently:

- Recommendations for Multi-transactor Designs
- MIPI DSI Transactor
- Video Transactor
- HDMI2_Sink Transactor
- DP Sink Transactor
- UART Transactor
- UART Transactor Bridging Feature (run xterm over UART TCP/IP server)
- JTAG T32 Transactor
- zRci-Based Testbench
- Transactor Callbacks for Pre/Post Checkpoint

This section also explains zRci-Based Testbench and Transactor Callbacks for Pre/Post Checkpoint.

The MIPI DSI, Video, DP Sink, and HDMI2 Sink transactors use a common VirtualExternalDisplay-based solution described in subsequent sections. For details about the checkpointing feature, see *ZeBu User Guide*.

Recommendations for Multi-transactor Designs

You must consider the following recommendations for multi-transactor designs:

Creation of New Transactor Objects in C++ Testbench

It is mandatory that new transactor objects are created (constructed) after `ZEMI3Manager::init()` and before `ZEMI3Manager::start()` methods:

```
m_zemi3->init();
…
createUartModules( "top.uart_driver_0", _uart1Mode, "top.uart_driver_1",
 _uart2Mode );
createJtagModule( "top.jtagXtor0" );
createUfsModules( "xtor_ufs_host_svs", "xtor_ufs_device_svs",
 "top.m_ufshostmonitor", "top.monitor" );
…
m_zemi3->start();
```

Hardware Access from the C++ Testbench

Make sure there is no access to transactor hardware prior to the clocks activation (`ZEMI3Manager::start()`):

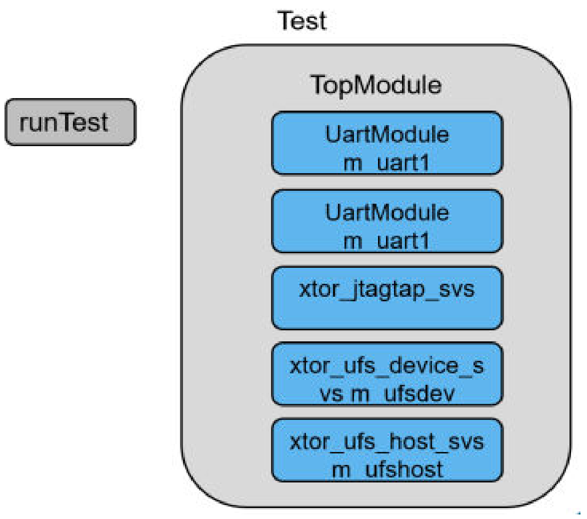
```
m_zemi3->start();
…
uartRunUntilReset();
jtagRunUntilReset();
if( !initUartModules( _rtsEnable ) ) return;

initJtagModule();
initUfsModules();
```

Prepare and Resume from a Checkpoint (C++ Testbench)

It is mandatory to evaluate `callBack_beforeCkpt` and `callBack_afterCkpt` callback functions for the `ZEBU::Board::checkpoint()` method when performing a checkpoint. Every transactor in the design must execute the `prepareCheckpoint()` method while being in the `callBack_beforeCkpt` callback and the `resumeCheckpoint()` method while being in the `callBack_afterCkpt` callback accordingly. The example testbench structure is shown in the following figure:

Figure 1   Example for Multi-transactor C++ Testbench Structure



Testbench (Test class) Scope

```
unsigned int  preCheckPoint ( void *_ptr ) {
    TOP_MODULE::TopModule *topPtr =  static_cast< TOP_MODULE::TopModule*
 >(_ptr);
    if( topPtr ) {
        topPtr-> prepareCheckpoint ();
    }
    std::cout << " End PRE checkpoint callback    " << std::endl ;
    return 1;
}

unsigned int  postCheckPoint ( void *_ptr, int _status ) {
   TOP_MODULE::TopModule *topPtr =  static_cast< TOP_MODULE::TopModule*
 >(_ptr) ;
   if( topPtr ) {
        topPtr-> resumeCheckpoint() ;
   }
   std::cout << " End POST checkpoint callback    " << std::endl ;
   return 1;
}

m_topModule->zebuCheckpoint( "CP1", &cpStatus, &preCheckPoint,
 &postCheckPoint, m_topModule.get(), true );
```

Transactor Wrapper (TopModule Class) Scope

```
void TopModule::prepareCheckpoint( void ) {
// UART
m_uart1 -> prepareCheckpoint ();
m_uart2 -> prepareCheckpoint ();
// JTAG
m_jtag -> prepareCheckpoint ();
// UFS
m_ufshost -> prepareCheckpoint ();
m_ufsdev -> prepareCheckpoint ();
}

void TopModule::resumeCheckpoint( void ) {
// UART
m_uart1 -> resumeCheckpoint();
m_uart2 -> resumeCheckpoint();
// JTAG
m_jtag -> resumeCheckpoint();
// UFS
m_ufshost -> resumeCheckpoint();
m_ufsdev -> resumeCheckpoint();
}

bool TopModule::zebuCheckpoint( const char *_checkpointDirectory,
                                  int *_checkpointStatus,
                                  unsigned int (*_callBack_beforeCkpt)
 (void *context), unsigned int (*_callBack_afterCkpt) (void *context, int
 checkpointStatus), void *_context, bool _suffix ) {

m_board->checkpoint( _checkpointDirectory, _checkpointStatus,
 _callBack_beforeCkpt, _callBack_afterCkpt, _context, _suffix );
}
```

Prepare and Resume from a Checkpoint (zRci Testbench)

It is mandatory to define additional `preCP` and `postCP` **zRci** commands (`zRci_command`) to handle `prepareCheckpoint()` and `resumeCheckpoint()` transactor methods when performing a checkpoint from the **zRci** Tcl flow.

zrci_run.tcl Scope

```
# performs DMTCP checkpoint if enabled
if {$dmtcp_enable == "dmtcp_en"} {
  puts "Performing SAVE STATE"
  testbench -command preCP 1
  checkpoint -fullsave CP1
  puts "Restoring the DMTCP_TEST"
  testbench -command postCP 1
}
```

C++ Testbench Scope

```
extern "C" std::string zRci_command(const std::string& key, const
 std::string& value) {
  if (key == "preCP") {
    printf("================================\n");
    printf("preCP called\n");
    printf("================================\n");

    // UART
m_uart1 -> prepareCheckpoint ();
m_uart2 -> prepareCheckpoint ();
// JTAG
m_jtag -> prepareCheckpoint ();
// UFS
m_ufshost -> prepareCheckpoint ();
m_ufsdev -> prepareCheckpoint ();
  }
  else if (key == "postCP") {
    printf("================================\n");
    printf("postCP called\n");
    printf("================================\n");
    // UART
m_uart1 -> resumeCheckpoint();
m_uart2 -> resumeCheckpoint();
// JTAG
m_jtag -> resumeCheckpoint();
// UFS
m_ufshost -> resumeCheckpoint();
m_ufsdev -> resumeCheckpoint();
  }
…
}
```

Note:

`"testbench -command postCP 1"` must be called again after restarting from a checkpoint.

You can call the `prepareCheckpoint()` and `resumeCheckpoint()` APIs using the **zRci** flow by defining the following callback functions and then calling xtor APIs from them:

```
void* zRci_pre_save (const ZRCI::TbOpts&);
void* zRci_post_save (const ZRCI::TbOpts&);

Example:

void* zRci_pre_save (const ZRCI::TbOpts&) {
  Xtor->prepareCheckpoint();
  return NULL;
}

void* zRci_post_save (const ZRCI::TbOpts&) {
  Xtor->resumeCheckpoint();
  return NULL;
}
```

The `zRci_pre_save()` and `zRci_post_save()` callback functions are called automatically when `checkpoint -fullsave` is executed.

MIPI DSI Transactor

The MIPI DSI transactor supports a built-in GTK-based virtual display that displays the images received on the DSI interface. With DMTCP, this built-in GTK is not supported. To provide the same level of support, the MIPI DSI transactor supports an external GTK-based Virtual Display (**VirtualExtDisplay**).

The **VirtualExtDisplay** interface is shipped separately and should be installed in `ZEBU_IP_ROOT`. Also, the testbench should link the **VirtualExtDisplay** library at runtime `-lVirtualExtDisplay`.

The GTK display GUI communicates to the MIPI DSI transactor at runtime through TCP-IP socket communication. To launch the GTK Display GUI, open the **VirtualExtDisplayApp** utility, which is located in the `$ZEBU_IP_ROOT/bin/` directory. **VirtualExtDisplayApp** runs as client and MIPI DSI transactor runs as server for the TCP-IP communication. To support the DMTCP checkpointing, the testbench must perform the following steps:

1. Disconnect DSI from **VirtualExtDisplayApp**
2. Call `board::checkpoint ()` to save HW/SW state
3. Reconnect DSI again to **VirtualExtDisplayApp**

Following are the APIs available in the MIPI DSI transactor:

- connectExternalDisplay ()
- disconnectExternalDisplay ()
- reconnectExternalDisplay()
- destroyExternalDisplay ()
- clearExternalDisplay ()

connectExternalDisplay ()

The `connectExternalDisplay ()` method starts the TCP-IP communication with the **VirtualExtDisplayApp**. After the testcase starts, image data is sent to **VirtualExtDisplay** and the frame are shown in the GTK GUI.

```
bool connectExternalDisplay (uint64_t portNum, uint64_t timeout)
```

Where,

- `portNum` is the TCP-IP port number for the server to start connection.
- **VirtualExtDisplay** must also be started using the same port number.

  ```
  $ZEBU_IP_ROOT/bin/VirtualExtDisplayApp -n "DSI" -a "localhost"  -p
   5000 -l 480 -w 640 -v v
  ```

  where,

  - `-p`: Specifies the port number to connect
  - `-a`: Specifies the TCP IP address
  - `-l`: Specifies the GTK display height
  - `-w`: Specifies the GTK display width
  - `-v`: Creates the verbose log

disconnectExternalDisplay ()

The API is used to disconnect the TCP-IP connection to the **VirtualExtDisplayApp**. Connection can be re-established after checkpoint using the `connectExternalDisplay ()` call.

reconnectExternalDisplay()

This API is used to reconnect the TCP-IP connection, when it is disconnected while using the DMTCP feature.

destroyExternalDisplay ()

`destroyExternalDisplay ()` sends a command to **VirtualExternalDisplayApp** to close the GTK Window. Connection must be established for this command to be successful.

clearExternalDisplay ()

`clearExternalDisplay ()` clears the content on the **VirtualDisplay** screen. This API is only available with the MIPI DSI transactor.

Example Usage

```
display = new (DSI);
display->init(board, "xtor_mipi_dsi_device_svs.u_DSI_driver");
```

```
// Connect to external display APP -------
display->connectExternalDisplay(5000,5000);

// Setting Xtor
display->setLog("dsi_debug.log",true);
display->setDebugLevel(1);
display->setWidth(width);
display->setHeight(height);
display->setMClkFreq(35.9);
display->setEnableLanes(4);
display->config(VIDEO_MODE);
// Start receiving for 2 frames
display->start (2) ;

while (!display->isHalted()) {
  display->serviceLoop();
}
// Disconnect to the VirtualExtDisplay
display->disconnectExternalDisplay();
// Create checkpoint
board->checkpoint("CP2", &ckptStatus, pre*, post*);
// Reconnect again
display->connectExternalDisplay(5000,5000);
// Restart frame reception
display->start (10) ;

while (!display->isHalted()) {
  display->serviceLoop();
}
```

State is restored immediately after the point of checkpoint creation.

```
% ./CP2.0/dmtcp_restart_script_zebu.sh
```

Video Transactor

The Video transactor uses the same approach as the MIPI DSI transactor and has the support to connect to the **virtualExtDisplayApp** utility using the TCP-IP socket communication. It provides the following APIs:

- connectExternalDisplay ()
- disconnectExternalDisplay ()
- destroyExternalDisplay ()

Example:

```
// Connect to external display APP -------
display->connectExternalDisplay(5000,5000);
```

```
    // Setting Xtor
#ifdef EXTSYNC
    display->setMode (modeYCrCb_422_10);
    display->setInterlace(false);
    display->setVSyncPolarity(false);
    display->setHSyncPolarity(false);
    display->setValidPolarity(true);
    display->setFieldPolarity(true);
#else
    display->setMode (modeYCrCb_656);
    display->setInterlace (false);
#endif
    display->setChannelCount (channel_count);
    display->setWidth(width);
    display->setHeight(height);
    display->setDebugLevel(1);
    display->config();
#ifdef ZEBU_VISUAL
    display->zoomOutVisual(50);
    display->rotateVisual(rotate90);
#endif

    ctxt.board      = board;
    ctxt.display    = display;
    ctxt.ptbEnd     = &tbEnd;

    // Register context for TB resources release
    tbRelease(&ctxt);

    // Start GTK main loop
    gtk_idle_add(gtk_idle_fun,(gpointer)&ctxt);

    // Start GTK handling thread
    if (pthread_create(&thread, NULL, gtkthread, &ctxt)) {
      perror("Could not start display thread\n");
      ret = 1;
    }

    // Start transactor for specified number of frames

 printf("*********************************************************\n");
    printf(" VIDEO start for %d Frames  \n",nbFrames);

 printf("*********************************************************\n");

    display->openDumpFile("Video_dump.log");

#ifdef RUN_DMTCP
  int ckptStatus = 0;
  display->start(4);
  while (!display->isHalted()) {
#ifdef USE_ZEBU_SLOOP
    board->serviceLoop();
#else
    display->videoServiceLoop();
#endif
  }
  display->disconnectExternalDisplay();
  board->checkpoint("CP1", &ckptStatus, NULL, NULL, NULL);
  display->connectExternalDisplay(5000,5000);
  display->start(6);
#else
  display->start(nbFrames);
#endif

    while (!display->isHalted()) {
#ifdef USE_ZEBU_SLOOP
      board->serviceLoop();
#else
      display->videoServiceLoop();
#endif
    }
```

HDMI2_Sink Transactor

The HDMI2 transactor uses the same approach as the MIPI DSI transactor and has the support to connect to the **virtualExtDisplayApp** utility using TCP-IP socket communication. It provides three similar APIs:

- connectExternalDisplay ()
- disconnectExternalDisplay ()
- destroyExternalDisplay ()

Example: run-> checkpoint -> continue run

```
cout<<"connecting external display";
hdmi2_rx->connectExternalDisplay(5000,5000);

cout<<"Starting Configuring Source Xactor ...\n";

hdmi2_tx->setDebugLevel(1);
zHDMI2_Source_Config *cfg_s;
cfg_s = new zHDMI2_Source_Config ();

cfg_s->sim_fast_mode              = true;
cfg_s->video_transmission      = STREAMING; //COLORBAR ;  //
cfg_s->mode                     = 1;
cfg_s->vic                      = vic;

hdmi2_tx->config(cfg_s);
cout<<"Configuration done"<<endl;
```

```
    hdmi2_rx->runUntilReset();
    cout<<"Configuring Sink Xactor ...\n";
    hdmi2_rx->setDebugLevel(1);
    zHDMI2_Sink_Config *cfg;
    cfg = new zHDMI2_Sink_Config();

    cfg->audioFmt                  = LPCM ;
    cfg->SourceType                 = XTOR ;
  #ifdef HDMI_VERSION_2_1
    cfg->HdmiVersion                 = HDMI_VER_2_1 ;
  #endif
    hdmi2_rx->config(cfg);

    hdmi2_rx->setPktDumpEnable("pkt_dump.txt");
    hdmi2_rx->setVideoDumpEnable("vid_dump.txt");
    hdmi2_rx->setAudioDumpEnable("aud_dump.txt");

    // Wait for Hot plug
    hdmi2_tx->waitForHPD () ;

    // Run the xtor for 6 frames
    hdmi2_tx->runFor(6);

    HDMI2_xtor_bfm_status_type status;
    zHDMI_xtor_status  status_sink ;
    unsigned int m_count ;

    // check for the bfmSTatus
    // for frame done event
    do {
         status = hdmi2_tx->getBfmStatus() ;
         status_sink = hdmi2_rx->getXtorStatus();
  #ifdef RUN_DMTCP
         if(status_sink.frame_nb == 2)
           {
             if(status_sink.line_nb == 100)
             {
               if(saved==0)
                {
                 cout<<"Frame count ="<<status_sink.frame_nb<<"\n";
                 hdmi2_rx->disconnectExternalDisplay();
                 board->checkpoint("CP2", &ckptStatus, NULL, NULL, NULL);
                 hdmi2_rx->connectExternalDisplay(5000,5000);
                 saved=1;
                }
             }
    }
  #endif

    } while (status.N_FRAME_DONE == 0 && hdmi2_rx->isHalt()==0 ) ;
```

EXAMPLE: Restore

```
% ./CP1.0/dmtcp_restart_script_zebu.sh
```

Restore happens after line 101 in 2nd frame as shown in the preceding example.

DP Sink Transactor

The DP Sink Transactor uses the same approach as the MIPI DSI Transactor and has the support to connect to the **VirtualExtDisplayApp** utility using TCP-IP socket communication. It provides the following set of APIs:

- connectExternalDisplay ()
- disconnectExternalDisplay ()
- reconnectExternalDisplay()
- destroyExternalDisplay ()

Example

```
cout<<"connecting external display"<<endl;
dp_rx->connectExternalDisplay(5000,5000);

cout<<"Configuring Sink Xtor ...\n";
dp_rx->setDebugLevel(1);

printf("Waiting reset end\n");

dp_tx->runUntilReset();

dp_rx->setLog("xtor_dp_sink_svs_debug.log",true);
dp_rx->setLinkRate(ZEBU_IP::XTOR_DP_SINK_SVS::LINK_RATE_540);
dp_rx->setEnhancedFramingMode(ENHANCE_EN);
dp_rx->setLanes(TB_NUM_LANE);
dp_rx->setTuSize(64);
dp_rx->setVideoCode(VIC);
dp_rx->setNoAUXLinkTrainEnable(true,1);
dp_rx->setVidMode(COLOR_FORMAT_SINK,COLOR_DEPTH_SINK);
dp_rx->enableOperatingMode(sst_mode_en);
dp_rx->setVideoSyncParams();
dp_rx->setVideoBlankActiveParams();
dp_rx->setHotPlugDetect(true);
dp_rx->start(3);

cout<<"Configuring Source Xtor ...\n";

dp_tx->setDebugLevel(1);

zDP_SourceConfig *cfg;
cfg = new zDP_SourceConfig ();
```

```
cfg->sim_fast_mode            = true ;
cfg->enableEnhanceFraming     = ENHANCE_EN ;
cfg->num_lanes                = NUM_LANE   ;
cfg->link_rate                = LINK_RATE;
cfg->streamClkDivFactor       = CLK_DIV_FACTOR ;
cfg->link_training_type       = FAST_LINK_TRAINING ;
cfg->video_transmission       = STREAMING ;
cfg->video_mapping            = FORMAT_DEPTH_SRC ;
cfg->vic                      = VIC;

dp_tx->config(cfg);

dp_tx->setVideoStreamingFile("../../src/video_files/Image_RGB_8bpc_640_48
0_COLORBAR_PATTERN.rgb");
dp_rx->setMainLinkTextDataFile("RGB_8_640_480_text.log");
dp_rx->setMainLinkBinaryDataFile("RGB_8_640_480_binary.log");


cout<<"Configuration done"<<endl;

dp_tx->runClk(16*10*10) ;
// Wait for Hot plug
dp_tx->waitForHPD () ;

//Do link training
dp_tx->doLinkTraining( );

cout<<"Starting the video for 1 frame"<<endl;

// Run the xtor for 1 frame
dp_tx->runFor(3);

dp_xtor_bfm_status_type status;
unsigned int m_count ;
// check for the bfmSTatus for frame done event
do {
  status = dp_tx->getBfmStatus() ;
  m_count ++ ;
  dp_tx->runClk(100) ;
  if(dp_rx->getFrameCount()==1) {
    if(dp_rx->getLineNumber()==250) {
      if(saved==0) {
        cout<<"Frame Count "<<dp_rx->getFrameCount()<<endl;
        cout<<"Line Number "<<dp_rx->getLineNumber()<<endl;
        cout<<"Disconnecting external display"<<endl;
        dp_rx->disconnectExternalDisplay();
        board->checkpoint("CP1", &ckptStatus, &preCheckPoint,
 &postCheckPoint, &context);
        cout<<"Re-connecting external display"<<endl;
        dp_rx->reconnectExternalDisplay();
        saved=1;
      }
      }
    }
  } while (!dp_rx->isDone ()) ;
```

In above example, transactor saves the state at line number 250 of the second frame and would restore at the same point.

Restore can be done by launching the auto-generated script in CP1 directory in the run directory that is created by checkpointing.

```
./CP.0/dmtcp_restart_script_zebu.sh
```

UART Transactor

The UART transactor provides three different use models using the following API which allows you to operate the Transactor in the required mode:

```
bool setOpMode (UartOperationMode_t mode);
```

where, the default mode is the normal mode of operation

The following table lists different UART transactor modes

Table 1 UART Transactor Modes

Class	Details
`DefaultMode`	UART transactor communication is done using APIs.
`XTermMode`	UART transactor opens an `xterm` window where the transmitted data can be provided as an input on the `xterm` and sent to the interface and the received data is displayed on the terminal.
`ServerMode`	UART transactor opens a TCP-IP communication with an external client/server. `Client`: UART transactor acts as a client and connects to a server application. `Server`: UART transactor acts as a server and can connect to the remote client.

The `Uart XtermMode` cannot be used with DMTCP because of GUI limitation. Therefore, you must use `ServerMode` for the UART transactor to have the same functionality as the `Xterm` mode.

It is recommended to use `ServerMode` in `Client-Mode` as it allows you to keep the connection of the server active and connect/disconnect the UART transactor.

Perform the following steps for proper client/server connection and communication:

1. On a new terminal, start a listening service by using the following netcat command:

   ```
   nc -lk <port-number>
   ```

Where, `<port-number>` specifies the port number that waits for a connection.

1. Start the testbench using `dmtcp_launch`.
2. Before a checkpoint, testbench must disconnect the connection and re-establish it again after the checkpoint.

Example: `run-> checkpoint -> continue`

```
// Set operation mode to ServerMode
uart->setOpMode (ServerMode); //Has to be always called in the main
 thread only
// Configuration
// …
// Start the client
if (!uart->startClient( tcp_server, tcp_port)) {throw ("Could not start
 UART Server."); }
// Do some data transfer…
// For checkpoint, first close connection and then do checkpoint and
 reconnect
close = uart->closeConnection();
    if (close) {
      printf("Client closed successfully\n");
      fflush(stdout);
    }
// Create checkpoint..
// Restore connection
if (!uart->reConnect()) {
      printf("Couldn't reconnect\n");
    }
    else printf("Re-connections successful\n");
  }
//Continue...
```

Example: Restore

You can restore the HW/SW state, by launching the auto-generated script in the CP1 directory that is created by the checkpoint, as follows:

```
% ./CP1.0/dmtcp_restart_script_zebu.sh
```

UART Transactor Bridging Feature (run xterm over UART TCP/IP server)

UART transactor can be enabled in TCP/IP and Xterm modes. However, only the TCP/IP mode is supported with DMTCP checkpointing due to the current DMTCP limitations. If you still want to have a seamless Xterm experience with DMTCP, use a special bridging technique, which involves `zCheckpointBridge` and `zSubprocessProtector` applications running over the UART transactor in the TCP/IP server mode. These binaries are delivered along with ZeBu.

A typical UART bridging scenario with the **zRci** flow is explained in the following steps:

1. Spawn bridge and xterm processes as soon as UART transactor TCP/IP server starts:

   ```
   proc launch_bridge {} {
       global bridgepid
       set bridgepid [exec <absolute_path>/zCheckpointingBridge -history
    13002 13000 &]
       puts "Bridge Launched with (virtualized) PID : $bridgepid"
   }
   proc launch_xterm {} {
   global env
   global xtermpid
       set xtermpid [exec dmtcp_nocheckpoint
    <absolute_path>/zSubprocessProtector xterm -display
    $env(ORIG_DISPLAY) -e nc localhost 13002 &]
       puts "Xterm Launched with (virtualized) PID : $xtermpid"
   }
   launch_bridge
   launch_xterm
   ```

   where:

   - 13000 is a port to connect to UART transactor server
   - 13002 is a port to connect to `nc/xterm`

2. Terminate bridge and xterm processes when emulation is completed (use `zebu_exit` instead of `finish`):

   ```
   proc zebu_exit {} {
           upvar myusername username
           set xterm_id [exec pgrep -u$username -n xterm]
           set bridge_id [exec pgrep -u$username bridge]
           exec kill -9 $xterm_id $bridge_id
           finish
   }
   ```

3. Re-spawn `xterm` process on restart using `save_ckpt <name>` instead of `checkpoint -fullsave <name>`:

```
set DMTCP_AFTER_RESTART 2
proc save_ckpt {name} {
    global DMTCP_AFTER_RESTART
    run -disable all
    if {[expr [checkpoint -fullsave $name] == $DMTCP_AFTER_RESTART]} {
        launch_xterm
    }
}
```

JTAG T32 Transactor

The JTAG T32 transactor works with **Lauterbach debugger** window. JTAG T32 transactor runs in server mode whereas the **Lauterbach** window works in the client mode. To enable the DMTCP in the testbench, perform the following steps:

1. Close the **Lauterbach** window (client side).
2. Stop server using `close()` API on server side and the save the state.
3. During restoring the state, re-run server.
4. Start the `startserver` API.
5. Launch the **Lauterbach** again.

Example

```
xtor_jtag_t32_server* t32_server = new xtor_jtag_t32_svs(0);

t32_server->init(board, "t32_top.jtag"
                ,PV_SWD , "t32_top.swd"
                ,PV_APB , "t32_top.apb_master_U0"
                runtime);

// ... configurations

cerr << "#TB : Starting jtag server\n";

t32_server->startServer(SocketNumberJtag,
                        SocketNumberDAP0,
                        SocketNumberDAP1);

cerr << "#TB : Main Loop\n";
while(!exit_flag) {   }

t32_server->close();
```

```
int ckptStatus=0;

board->checkpoint("save_axi", &ckptStatus, NULL, NULL, NULL);

t32_server->startServer(SocketNumberJtag,
                        SocketNumberDAP0,
                        SocketNumberDAP1);

cerr <<"#TB : Main Loop 2 \n";

// ... continue
```

zRci-Based Testbench

In **zRci**-based testbenches, use the `zRci_command()` to `connect ()` /`reconnect`.

Example

```
extern "C" std::string zRci_command(const std::string& key, const
 std::string& value) {
  std::string ret_val_str = "UnknownCommand";
  printf("Processing command \"%s\"\n",key.c_str());
  if(key == "uartConnect") {
   if (!test->startClient( tcpServer, tcpPort)) {throw ("Could not start
 UART Server."); }
    ret_val_str = "OK" ;
} else if (key == "uartDisconnect") {
 int close = test->closeConnection();

 if(close) {
   ret_val_str = "OK" ;
 }
}
}
```

Transactor Callbacks for Pre/Post Checkpoint

Transactors consist of the following callback APIs for DMTCP checkpoint:

- `prepareCheckPoint ()`
- `resumeCheckPoint ()`

These callbacks allow the transactor to release the license at the point of checkpoint and acquire the license post checkpoint again.

To acquire the license, call these APIs from the callbacks and attach to `board::checkpoint`.

The following example shows the implementation of a DMTCP checkpoint using the above APIs:

```
unsigned int  preCheckPoint ( void *cntx)
{
   cout << "====================================" << endl ;
   cout << " Calling PRE checkpoint callback    " << endl ;
   cout << "====================================" << endl ;
   TbCtxt *ctxt =  (TbCtxt*) cntx ;
   if(ctxt != NULL) {
     ctxt-> master -> prepareCheckpoint () ;
     ctxt-> slave  -> prepareCheckpoint () ;
   }
   cout << " End PRE checkpoint callback    " << endl ;
   return 1 ;
}

unsigned int  postCheckPoint ( void *cntx, int status )
{
   cout << "====================================" << endl ;
   cout << " Calling POST checkpoint callback    " << endl ;
   cout << "====================================" << endl ;
   TbCtxt *ctxt =  (TbCtxt*) cntx ;
   if(ctxt != NULL) {
     ctxt-> master -> resumeCheckpoint() ;
     ctxt-> slave  -> resumeCheckpoint() ;
   }
   return 1  ;
}

//add the callbacks in the checkpoint call
board->checkpoint("CP1", &ckptStatus, &preCheckPoint,&postCheckPoint,
 &context);
```

Feedback

7

Checkpointing and Restore for Simulation Acceleration

The Checkpoint and Restart feature in Simulation Acceleration is a useful debug feature that provides you the ability to run the simulation to a specific point and save the state (checkpoint). Subsequently, you can run different tests from this saved state. During the run, you can save multiple checkpoints and restart the simulation from different checkpoints to analyze a problem. This helps avoid the cost of run until the points that have been verified.

The Simulation Acceleration support for the Checkpoint and Restart feature is implemented on top of the ZeBu DMTCP Checkpoint and Restart feature. For more information on the ZeBu use model, see the following chapters:

- Performing a Checkpoint
- Restarting Emulation from a Checkpoint

In Simulation Acceleration, the entire states of both the testbench and HW states are saved by using the ZeBu DMTCP mechanism.

For more information, see the following subsections:

- Prerequisites
- Use Model for Checkpoint and Restart in Simulation Acceleration
- Limitations of Checkpoint and Restart in Simulation Acceleration

Prerequisites

The Simulation Acceleration Checkpoint and Restart feature is only applicable to SystemVerilog designs.

Use Model for Checkpoint and Restart in Simulation Acceleration

Note:
For information about setting environment variables, refer to Post-Installation Tasks and Additional Variables for DMTCP.

The use model is as follows:

1. **Create a Checkpoint**: Identify a suitable point in the emulation runtime, where a checkpoint needs to be saved. Use the following command to create a checkpoint:

   ```
   dmtcp_launch <simv> -ucli -i <ucli.tcl> <simv arguments>
   ```

   The `ucli.tcl` file is created based on your requirements. Suppose, you want to perform two checkpoints, at 40 ns, and 70 ns respectively, and save the data in directories named `save1.0` and `save2.0` (suffix 0 is used for the first checkpoint). The following snippet shows an example of the `ucli.tcl` file:

   ```
   run 40ns
   save save1
   run 30ns
   save save2
   run
   ```

   If you restore the run from checkpoint `save1`, in the course of execution it creates a checkpoint `save2`, and the data is saved in a directory name `save1.0.save2.0` (checkpoints are hierarchical, it does not overwrite the `save2.0` directory).

2. **Restore a Saved Checkpoint**: To restore the emulation from a specific checkpoint, navigate to the directory where the original emulation run was executed and then run the following script in the checkpoint directory:

   ```
   ./<save_directory.int>/dmtcp_restart_script_zebu.sh
   ```

   This restarts the execution of the testbench from the exact location where it was saved.

   Note:
   The `rundir` can be relocated. For more information about relocating the `rundir` for this use model, refer to the chapters Checkpoint Sharing Use Models and LCA Features.

3. **Restoring Session in Simulation Acceleration**: After restore, Simulation Acceleration continues to execute the command script provided in the first run. This is useful in following scenarios:

   - When the emulation run is for a long duration, and you want to save the state at an intermediate stage and then continue later
   - In batch mode for regressions. An example of the UCLI script is as follows:

     ```
     run 40ns
     save save1
     source cmd.tcl
     exit
     ```

     Where, the `cmd.tcl` file can be modified to execute different commands for different runs.

Alternatively, on restore, you can use the UCLI prompt to specify commands that need to be executed in the session. To use this approach, use the following option:

```
-simxl=ucliOnRestore
```

For example:

```
dmtcp_launch <simv> -ucli -i <ucli.tcl> -simxl=ucliOnRestore <simv
 arguments>
```

If the first run is invoked in an interactive mode (without input script), the restore session starts with the UCLI prompt.

4. **Generating Output**: To generate the output log, use the `-l` option of `simv` which allows redirecting the output to a file.

   However, to redirect the output to the same file or a different file in the Restore run session, it is mandatory to redirect the output in the Checkpoint run session.

   When outputting waveforms, you must provide the UCLI commands again to open the waveform file and start output. It is possible to use the same name as the file in the save session, however the previous waveform file is overwritten.

Limitations of Checkpoint and Restart in Simulation Acceleration

- **Multiprocess testbench is not supported**: The `-ucli2Proc` does not work in this mode.

Feedback

8

Performance

This section discusses the following topics:

Compression

This section discusses the following topics:

7 ZIP Support

7-zip compression engine provides the most optimal compression and decompression speed and ratio when used with DMTCP for checkpointing the software state of emulation (`.dmtcp` file)

The 7-zip utility is delivered with Zebu and can be found in the installation directory at `$ZEBU_ROOT/thirdparty/dmtcp/7Zip`:

- `7zzs: ready-to-use binary`
- `./src: original source files`

To use the 7-zip utility as the default compression/decompression engine for DMTCP, add the 7-zip binary to the `PATH` environment variable:

```
setenv PATH $ZEBU_ROOT/thirdparty/dmtcp/7Zip:$PATH
```

When DMTCP is installed using the recommended `install_dmtcp.sh` script, the `dmtcp_settings.sh(csh)` scripts contain the command to add 7-zip binary to the `PATH` environment variable.

Refer to `dmtcp_launch --help` to check how to use 7-zip compression tool with DMTCP.

The `--7zmx <compression-level>` and `--7zmmt <threads>` parameters are optional for optimal 7-zip compression ratio and speed. The `--7zmx <compression-level>` parameter is set to 2 by default and the `--7zmmt <threads>` parameter is set to 1.

Converting fastState to hardwareState Between DMTCP Save and Restart

DMTCP allows saving a state of the test to restart in case of debug, or saving a booting time. Some users create a DMTCP checkpoint at every interval time. This allows the user to restart the test at any point and continue with the verification.

To restart from a checkpoint, it is necessary to convert the `fastState` to `hardwareState` during the restart process. Conversion of large designs take time and increase the restart time. You can reduce this conversion time from first restart by converting between save and restart steps.

The `convert_ckpt_hw_state.sh` script simplifies converting `fastState` to `hardwareState` between save and restart steps. Version 1.1 of this script is available at: `$ZEBU_ROOT/thirdparty/dmtcp/scripts`.

The following table lists all available options:

Option	**Description**
`-d <path to directory with checkpoints>`	Path to a directory with checkpoints.
`-c <path to checkpoint directory>`	Path to the checkpoint directory.
`-z <path to zcui.work directory>`	[optional] Path to the `zcui.work` directory.
`-l <log_file path>`	Path to log file.
`-h`	Displays help.

- The script checks whether the `hardwareState` exists. If yes, then the script does not perform a conversion.
- If the `-c` option is used, then the script converts the `fastState` for the provided checkpoint.
- If the `-d` option is used, then the script finds all completed checkpoints in the folder and perform the conversion as required.

- If the `-z` option is not provided, then the script automatically gets a `zcui.work` path from the `dmtcp_restart_script_zebu.sh` script.

  The script should be used between save and restart, and it should not perform a conversion on restart.

Forked Checkpointing

DMTCP forked checkpointing (`dmtcp_launch --fork`) is used to run checkpointing of SW part in a background process. It ensures checkpointing does not block the emulation flow and allows the software execution to proceed.

Certain issues may be observed when a user tries to run the restart script before checkpointing is complete. To avoid potential issues, user can check the status of checkpoint (forked) in one of the following ways:

1. `$ZEBU_ROOT/thirdparty/dmtcp/scripts/check_dmtcp_ckpt_status.sh` script:

   Example usage:

   `check_dmtcp_ckpt_status.sh -c <path to checkpoint directory>` displays the status of a checkpoint.

   Refer to `check_dmtcp_ckpt_status.sh -h` to display complete help information.

2. `$ZEBU_ROOT/thirdparty/dmtcp/scripts/zRci/dmtcp.tcl` helper Tcl procedures (dmtcp namespace) to run from zRci command line:

   - `check_ckpt_status`:

     ```
     # Procedure to check DMTCP checkpoint status
      # ckpt_path - path to a DMTCP checkpoint
      #
      # Returns:
      # 0 - The checkpoint is saved
      # 1 - The checkpoint is not saved
      # 2 - In case of error
     ```

   - `check_all_ckpts_status`:

     ```
     # Procedure to check DMTCP checkpoints status
      # db_path - zRci DB name used to store DMTCP
     checkpoints
      #
      # Returns:
       # 0 - All DMTCP checkpoints located in provided DB
     are saved
       # 1 - Some DMTCP checkpoints located in provided
     DB are not saved
       # 2 - In case of an error
     ```

- `wait_until_forked_ckpt_done`:

  ```
  # Procedure to wait until a DMTCP checkpoint is saved.
   # ckpt_path - path to a DMTCP checkpoint
   # timeout - Timeout in minutes. The default
  timeout period is 1 minute.
   # An infinite timeout may be set with the
  value <= 0.
   #
   # Returns:
   # 0 - The checkpoint is saved
   # 2 - In case of an error
   # 3 - In case of a timeout
  ```

- `wait_until_all_forked_ckpts_done`:

  ```
  # Procedure to wait until all DMTCP checkpoints located
  in provided DB are saved.
   # db_path - zRci DB name used to store DMTCP
  checkpoints
   # timeout - Timeout in minutes. The default
  timeout period is 1 minute.
   # An infinite timeout may be set with the
  value <= 0.
   #
   # Returns:
  ```

9

Checkpoint Sharing Use Models

This chapter discusses the following topics:

- Basic Flow
- Pathvirt Plugin

Basic Flow

This section discusses the following topics:

Introduction

This section discusses the following two use models:

- Use Model 1: User1 saves the checkpoint and also restores it on same/different host.
- Use Model 2: User1 saves the checkpoint but shares it with user 2 who in turn restores the checkpoint on same/different host.

Note:

DMTCP package must be installed on every runtime host you plan to run DMTCP on unless the hosts have same configurations.

A detailed basic checkpoint sharing example is available in Appendix A: Basic Checkpoint Sharing Use Model Example.

Step 1: Checkpoint Save

After running for some clock cycles, you may want to save the checkpoint.

On the **zRci** prompt, enter: `checkpoint -fullsave at1`.

Where, `at1` indicates the name of the your checkpoint directory.

In this example, the runtime directory as mentioned in **zRci** is `ZRCI_1`. This creates a folder in your runtime directory (`ZRCI_1`): at1.

Feedback

Depending on the size of your design and the number of ZRMs, this step takes some time to complete. You can exit from **zRci** after the checkpoint is saved.

Use Model 1

Perform the following additional steps for use model 1, in continuation with the command steps.

Step 2: Checkpoint Restore

You can either restore the checkpoint from same terminal, or from a new terminal; either using the same host, or a different host.

To restore the checkpoint:

1. Take a new terminal and login to the host.
2. Set the following environment variables:
    - `ZEBU_PHYSICAL_LOCATION`

Note:

Always restore from your runtime directory. Otherwise, the restore process will error out. This step will take some time, but it will restore at the point you had initially saved for.

Use Model 2

Perform the following additional steps for use model 1, in continuation with the command steps.

Note:

The runtime directory is assumed as `zebu_run`.

In this use case, User1 will share the entire runtime directory with User2 so that User2 can access the checkpoint saved by User1.

Step 3: File Permissions

User1 must make sure that the `zrci_command.log` has permissions set to `777`.

If any monitor/transactor log was present but was not closed while saving the checkpoint, then set the file permission of these log files to `777`.

Step 4: Checkpoint Restore

User 2 must perform the following steps to restore the checkpoint:

1. Login to the host.
2. Navigate to the copied `rundir` area folder.

   For example: `myruntimedir`
3. Set the following environment variables:
   - `ZEBU_PHYSICAL_LOCATION`

Note:

Always restore from your runtime directory. Otherwise, the restore process errors out. Although this step takes some time, it restores at the point you had initially saved for.

Pathvirt Plugin

Pathvirt is a DMTCP plugin that virtualizes access to the file system paths for programs run under DMTCP. It allows path access to be transparently redirected to a different location in the file system on DMTCP restart while moving checkpoints between directories or hosts.

To enable pathvirt support, run `dmtcp_launch` with `--pathvirt` option.

Pathvirt plugin operates based on path prefixes. By default, 4 mandatory paths are supported for virtualization in `dmtcp_restart_scrtipt_zebu.sh` restart script using `ORIGINAL_PATHS` and `NEW_PATHS` variables (the order is important):

1. P1 - relative path to the checkpoint directory
2. P2 - absolute path to the `zCui.work` database
3. P3 - absolute path to the checkpoint directory
4. P4 – emulation `rundir` directory

By default, `NEW_PATHS` values are generated at runtime based on the current restart script location. If the checkpoint (or `rundir`/backend databases) is moved to another location, then `NEW_PATHS` variable can be manually modified in `dmtcp_restart_script_zebu.sh` to reflect new changes in the file system.

The following is a snippet of restart script representing the path mapping generated by default:

```
# original paths
        ORIGINAL_PATHS=(
            # P1 - relative path to the checkpoint dir
            "./CP1.0"
            # P2 - absolute path to the zCui.work
            "${ZEBU_ZCUI_ORG}"
            # P3 - absolute path to the checkpoint dir
            "/username/projectname/zebu/rundir-VCS2021.09_dmtcp/CP1.0"
            # P4 = rundir
            "/username/projectname/zebu/rundir-VCS2021.09_dmtcp"
        )

        # new paths after checkpoint was moved, rules are
        NEW_PATHS=(
            # P1 - replace relative path to the checkpoint dir
            "./${SCRIPT_DIR##*/}"    # this is current name of checkpoint
            # P2 - replace ZeBu zCui workdir
            "${WORK_PATH_NEW}"       # zCui.work
            # P3 - replace checkpoint dir
            "${SCRIPT_DIR}"
            # P4 - replace checkpointed rundir
            "${MPWD}"
        )
```

Additional directories for path virtualization can be appended to the default directories by `DMTCP_ORIGINAL_PATH_PREFIX` and `DMTCP_NEW_PATH_PREFIX` variables in the restart script (`dmtcp_restart_script_zebu.sh`), using colon-delimited lists.

For example:

```
DMTCP_ORIGINAL_PATH_PREFIX=/home/user1/bin1:/home/user1/lib1
DMTCP_NEW_PATH_PREFIX=/home/user2/bin2:/home/user2/lib2
```

Path remapping is applied while executing the restart script.

Note:

The number and order of entries in `ORIGINAL_PATHS` and `NEW_PATHS` variables must be the same. Same rule applies to `DMTCP_ORIGINAL_PATH_PREFIX` and `DMTCP_NEW_PATH_PREFIX` variables.

Figure 2 Default path virtualization template

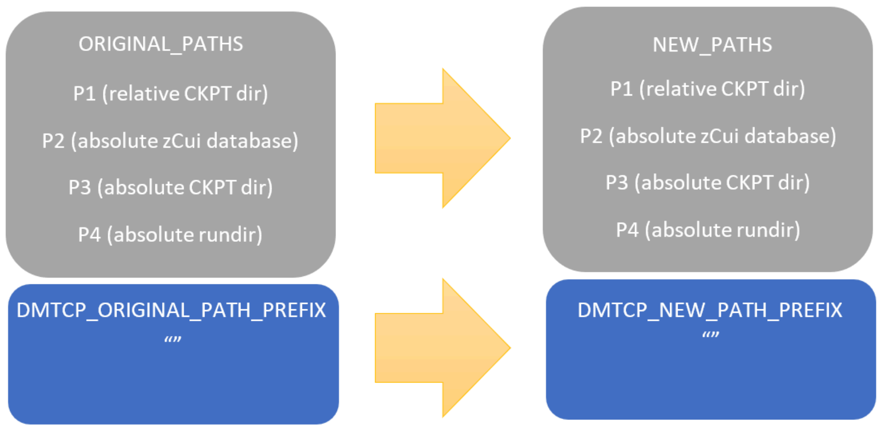


Figure 3 Typical path virtualization for checkpoint sharing

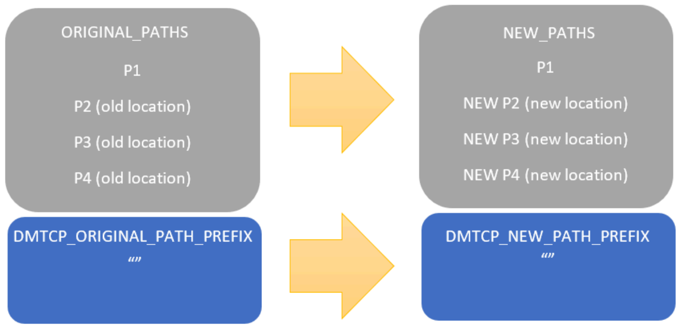
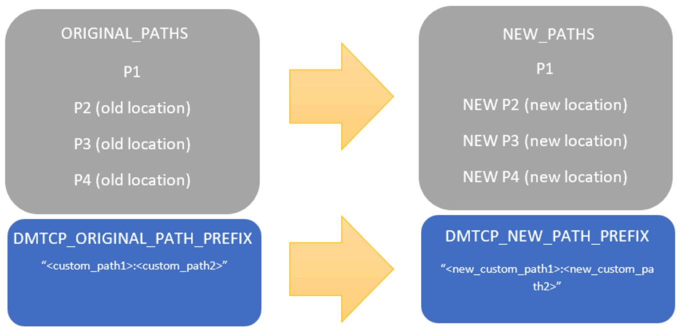


Figure 4 Custom path virtualization for checkpoint sharing



If the user does not provide `DMTCP_ORIGINAL_PATH_PREFIX`/`DMTCP_NEW_PATH_PREFIX` values on restart, or keeps the default values of `ORIGINAL_PATHS`/`NEW_PATHS` while project files location remains the same, then the pathvirt plugin does nothing, and the program is restarted from the checkpoint as usual.

If the user provides changes, then the pathvirt plugin is activated and starts performing path translation. When a program uses a path that is prefixed with an entry from provided variables in a call to `libc`, prefixes in the `NEW_PATHS`/`DMTCP_NEW_PATH_PREFIX` options dynamically replace corresponding prefixes in `ORIGINAL_PATHS`/ `DMTCP_ORIGINAL_PATH_PREFIX` options.

For example:

```
    open("/home/user1/bin1/ls") => open("/home/user2/bin2/ls")
    open("/home/user1/lib1/libc.so.6") =>
 open("/home/user2/lib2/libc.so.6")
    open("/not/registered/prefix.txt") =>
 open("/not/registered/prefix.txt")
```

Observe that the pathvirt plugin translates pair entries one-by-one to avoid unwanted path substitutions. When manually changing `DMTCP_ORIGINAL_PATH_PREFIX`/`DMTCP_NEW_PATH_PREFIX` and `ORIGINAL_PATHS`/`NEW_PATHS` values, keep the entries from the longest path to the shortest.

Following is an example of an incorrect order:

```
/A -> /D
/A/B -> /C/Z
```

In this case, `/A/B -> /C/Z` remapping will not happen, since after first entry pair remapping `/A/B` will become `/D/B`.

The correct entry pair order should be:

```
/A/B -> /C/Z
/A -> /D
```

Caution:

To enable pathvirt plugin functionality, launch `dmtcp_launch` with `--pathvirt` option.

Note:

The user can force the pathvirt plugin to turn-off on restart by setting the `DMTCP_DISABLE_PATHVIRT` environment variable.

For more information about pathvirt plugin usage, see `dmtcp_launch --help` section.

10 Limitations

The limitations of DMTCP are as follows:

- At restart, the DMTCP tool reopens any open file in its original location. Any data written after the checkpoint might be overwritten.
- The checkpointing and waveform capture features cannot be used simultaneously. The user must stop waveform capture before running checkpoint save and use a new dump file at restart.
- Any socket connections must be closed/disconnected before checkpoint creation.
- Spaces are not allowed within environment variables.
- The checkpointing feature does not work with a multiprocess testbench except for Hybrid/Virtualization use case. For Hybrid/Virtualization solutions that allow multiple external processes connected to the same **zServer**, new APIs (`beginOfCheckpoint()`, `endOfCheckpoint()`, `endOfRestore`) must be evaluated to prepare **zServer** for upcoming checkpoint to be created.
- When restarting emulation from a checkpoint, it is recommended to restart from the same run directory because this is where the checkpointed process was running and any file opened with a relative path uses this directory as a base. This limitation can be avoided using `pathvirt` plugin, or Podman container virtualization mechanisms.
- Starting the emulation multiple times from the same checkpoint at the same time in the same directory does not work because each run reuses the same files. Using another directory in such scenarios is not guaranteed to work as expected.
- When using the ZeBu runtime checkpointing feature, or **zRscManager Suspend**/**Resume**, make sure that the environment variable, `PATH` does not exceed 4KB for preventing any spurious impact.
- The runtime checkpointing feature supports only one user process in addition to the ZeBu runtime processes, such as **zRci** and **zServer**.
- For C++ testbench, `checkpoint()` must be called after `Board::init()`.
- Restart the emulation on the same OS used for the checkpoint. Otherwise, a DMTCP error is reported as follows:

```
***Error: vdso/vvar order was different during ckpt.
```

- DMTCP must be compiled from a host running the same OS as the runtime hosts.
- GUI-based and interactive processes (`vim`, `xterm`) are not supported under DMTCP, indicating that `DISPLAY` envvar is unset by DMTCP code. To spawn the GUI process outside of DMTCP hood using `dmtcp_nocheckpoint` command, use `ORIG_DISPLAY` envvar to pass to `DISPLAY` during the process launch. For example, `dmtcp_nocheckpoint <gui_process> DISPLAY=$ORIG_DISPLAY`.
- Hosts hardware and ZeBu PCIe boards setup must match for the checkpoint and restart.
- Using `zRci --attach <pid>` and DMTCP simultaneously is not supported.
- When starting an emulation under DMTCP, if the `PATH` environment variable is bigger than 4KB, a fatal error is generated.
- When restarting from a previous checkpoint, the following warning is displayed:

  ```
  WARNING : PATH Size is bigger than 4K becomes a fatal ERROR.
  ```

  Note:

  The check is localized in a script generated when the checkpoint is done. If the advanced mode environment variable, `ZEBU_DEBUG_CHECKPOINT_NO_KERNEL_CHECK` is set, the error in the script becomes a Warning.)
- If you want to run from another directory, you must copy all the `workdir` content into the new location and ensure that the testbench does not write files using the absolute path.
- If you are using DMTCP with RTL clocks and are using `setenv ZEBU_TXTOR_MODE 2` to stop time capture transactor, legacy Stimuli Capture and Replay technology is not supported.
- If you are using DMTCP and `setenv ZEBU_TXTOR_MODE 2` with Stimuli Replay and Capture, replay can only be done with cycles and not with time.
- It is not recommended to run checkpoint save while zPrd/sniffer is enabled. It is recommended to stop zPrd/sniffer before save and enable it after checkpoint save.
- The DMTCP installation can fail if a host configuration does not fulfill DMTCP installation requirements.

11

LCA Features

This section discusses DMTCP features that are released as Limited Customer Availability (LCA):

- Docker Support for PATH Virtualization
- Podman Container Support

Note:

These LCA features are related to Checkpoint Sharing Use Models.

Docker Support for PATH Virtualization

Why Docker?

- Docker can resolve the DMTCP checkpointing limitations for multi-user and multi-host support.
- DMTCP checkpoints are related to the file system:
  - every input/output files used during emulation (including zCui compilation database) must be in the original file path during a restart.
  - different users may either overwrite files, or multiple users may simultaneously edit the same file.
- Contenerizing ZeBu emulation provides a stable OS layer where all files are mounted in the same directory structure.

Using ZeBu with DMTCP in a Docker container requires a few important steps such as mounting right folders or configuring necessary environment settings.

This chapter discusses the necessary information for using Docker to run ZeBu with DMTCP in a hermetic Linux OS.

This chapter contains the following topics:

- Docker Versions
- Using Docker in Synopsys ZeBu
- FAQs

Note:
For information about using Podman, refer to Podman Container Support.

Docker Versions

The following two versions of Docker are available: Docker and Docker rootless. Both these versions can be used with DMTCP and Synopsys ZeBu. For more information, refer to Docker documentation.

The following table lists the most important differences between Docker and Docker rootless.

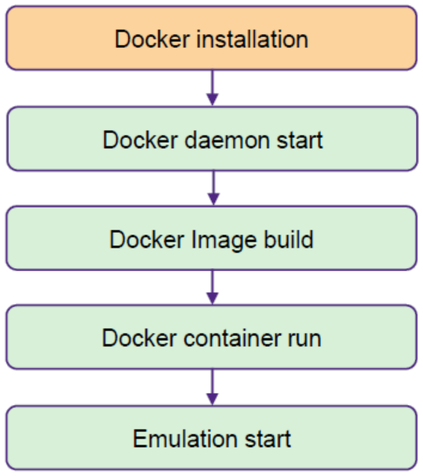
Table 2 Differences between Docker and Docker rootless

	Docker	**Docker Rootless**
Execution	As root	As user
Storage drivers' limitations	NA	Only the following are supported: • RHEL7/CentOS7 ◦ vfs • RHEL8/CentOS8 ◦ overlay2 ◦ fuse-overlays ◦ btrfs

Note:
Refer to the Docker documentation for information and limitations about Docker rootless.

Using Docker in Synopsys ZeBu

The following figure depicts the steps to use Docker with DMTCP and Synopsys ZeBu.



- **Docker installation**: This step involves installation using the Docker Engine installation steps.
- **Docker daemon start**: Docker rootless daemon is a user process, and must be started by the user.
- **Docker Image build**: This step involves creating an image using the docker build. Docker image is a template in which the container instance executes ZeBu emulation. Docker image can be reused by different users to run the Docker container.
- **Docker container run**: This step involves running the Docker container based on the prepared image.
- **Emulation start**: This step involves running the ZeBu emulation within the Docker container.

Preparing the Docker Image

Before executing any steps, copy all provided scripts from the `$ZEBU_ROOT/thirdparty/dmtcp/scripts/docker/` folder to your design folder.

1. Go to the folder containing the copy of all the provided template scripts and set `SNPS_DOCKER_IMAGE_NAME` variable with a name of the image.
2. If additional packages in the docker image need to be installed, apart from the minimal list of packages required by ZeBu, update the script as required.
3. Execute the following commands:

```
chmod 777 ./build_docker.sh
export SNPS_DOCKER_IMAGE_NAME=<YOUR_IMAGE_NAME>
./build_docker.sh
```

Running the Docker Container

Perform the following steps to run the Docker container:

1. Before running the Docker, source environmental settings from ZeBu and set right `ZEBU_SYSTEM_DIR` which corresponds to the ZeBu configuration which you are going to use.
2. Set these variables:
   - `SNPS_DOCKER_IMAGE_NAME`: name of the image
   - `SNPS_SHARED_FOLDER`: a path to folder to share with Docker
   - `DOCKER_MOUNT_FOLDER`: a path to folder in Docker which will be used to mount SNPS_SHARED_FOLDER
   - `DOCKER_ROOTLESS`: set to `yes` if Docker rootless should be used
3. If additional packages in the docker image need to be installed, apart from the minimal list of packages required by ZeBu, update the script as required.
4. Run the following commands:

```
chmod 777 ./run_docker.sh
export SNPS_DOCKER_IMAGE_NAME=<YOUR_IMAGE_NAME>
export SNPS_SHARED_FOLDER=<YOUR_FOLDER>
export DOCKER_MOUNT_FOLDER=<FOLDER_IN_DOCKER>
./run_docker.sh
```

Note:

- `YOUR_FOLDER` should point to a folder you want to mount in the Docker Linux system. In case of errors while mounting your folder in the Docker, refer to Docker documentation.
- A custom design may require more Host folders to be mounted in the Docker. Therefore, it may be necessary to update the `mount_rw` variable in `run_docker.sh` script. Refer to the FAQs section for more information.

Using ZeBu with DMTCP in the Docker Container

Perform the following steps to use ZeBu with DMTCP in Docker container:

1. Configure `g++`.

   Prepare `setenv.sh` file in your design folder and copy the content of g++ Configuration to it.
2. Source the following file: `source ./setenv.sh`
3. Source the ZeBu environment settings: `source <PATH_TO_ZEBU>/zebu_env.sh`

4. Use `install_dmtcp.sh` script to compile DMTCP:

   ```
   $ZEBU_ROOT/thirdparty/dmtcp/scripts/install_dmtcp.sh -d -b
   ```

   This compilation script also creates the `dmtcp_settings.sh` script to source before using DMTCP.

5. Set the ZeBu configuration by exporting the following variables:

   ```
   export FILE_CONF=<configuration>
   export ZEBU_SYSTEM_DIR=<configuration>
   ```

   Every ZeBu system has its own configuration folder which is provided by `FILE_CONF` and `ZEBU_SYSTEM_DIR` variables. Replace `<configuration>` by the configuration being used for each variable.

6. Compile TB and run a test.

   If necessary, source any custom design settings and start the test.

7. Once Docker container is running, checkpoint save is done in a standard way as described in Performing a Checkpoint.

g++ Configuration

```
export PATH=<configuration>
export LD_LIBRARY_PATH=<configuration>
```

Replace `<configuration>` by the configuration being used for each variable.

FAQs

How to enable Docker rootless?

To enable Docker rootless, set `DOCKER_ROOTLESS="true"` before calling the `run_docker.sh` script.

How to mount more design folders in Docker?

1. Open `run_docker.sh` script and find the following lines in the script:

   ```
   #array of user paths mapped to Docker
   # format is ( ["user_path"]="path_in_docker"
   ["second_user_path"]="path_in_docker")
   declare -A user_paths=(
   ["$SNPS_SHARED_FOLDER"]="$DOCKER_MOUNT_FOLDER")
   ```

2. Add new folders to mount after the provided code:

   ```
   user_paths+=( ["<NEW_DESIGN_FOLDER>"]="<NEW_FOLDER_IN_DOCKER>" )
   ```

How to module load ZeBu and VDK on Docker?

1. Before using Docker, load and configure the host environment to use both ZeBu and VDK. All necessary environment variables are automatically passed to the Docker.
2. Open the `run_docker.sh` script and find the following lines:

   ```
   # list of directories to bind/mount into container as read/write, in
    the same path
   mount_rw=
   ```
3. Add all necessary host folders with ZeBu and VDK to bind/mount into container after the provided code:

   ```
   mount_rw="$mount_rw <ABSOLUTE_PATH_TO_FOLDER>"
   ```

What is the difference between rootful/rootless?

Rootless indicates that you do not require root privileges to execute it. Rootful means that it must be run as root. Although Docker requires root, there is a rootless version that can be run without root privileges.

How to pass specific folders to the container, what must be passed to be able to run ZeBu emulation with DMTCP checkpointing?

- ZeBu driver and folders such as `/zebulab` or `/zebu` that is required by ZeBu to run emulation is added to the `run_docker.sh`.
- Folders and files required by the Synopsys environment should be added to the script that is in ZeBu release. These files are different for each customer environment. To run ZeBu emulation in Synopsys environment, open `run_docker.sh` script and modify the following to mount/bind all necessary directories into the container as read/write :

  ```
  # list of directories to bind/mount into container as read/write, in
   the same path
  mount_rw=
  ```
- Files/folders specific to the project such as the project directory to mount can be added to `run_docker.sh` here:

  ```
  #array of user paths mapped to Docker
  # format is
   ( ["user_path"]="path_in_docker" ["second_user_path"]="path_in_docke
  r")
  declare -A user_paths=( ["$SNPS_SHARED_FOLDER"]="$DOCKER_MOUNT_FOLDER"
  [<DMTCP directory path>]="/DMTCP_install")
  ```

What is the significance and difference between daemon and daemon-less architecture?

Daemon is the main difference between Docker and Podman. Docker requires daemon (program in background) to create and run containers. Podman is daemon-less which means that it does not require any background program, thus it can be run under the user and do not require root.

What is Docker image?

Docker image is the OS layer template which is used by the container to execute programs. The build image is creating using this OS layer template and will be used by the container. Dockerfile contains commands to create the image. The image is built using docker build command which is wrapped by `build_docker.sh` script.

Will saved checkpoint be a part of Docker image?

The use case is to mount every required directory to the Docker. Any created/saved file/ checkpoint/database will be created inside the container, and will be on a physical disk that is mounted to the Docker container.

How to start emulation inside the container, does it require any extra steps?

Container that runs the image is an OS layer on top of Host OS with ZeBu hardware. All required files and scripts are mounted inside the container. Therefore, the procedure to start emulation within container is the same as without the container.

How to check that Docker is, or can be installed?

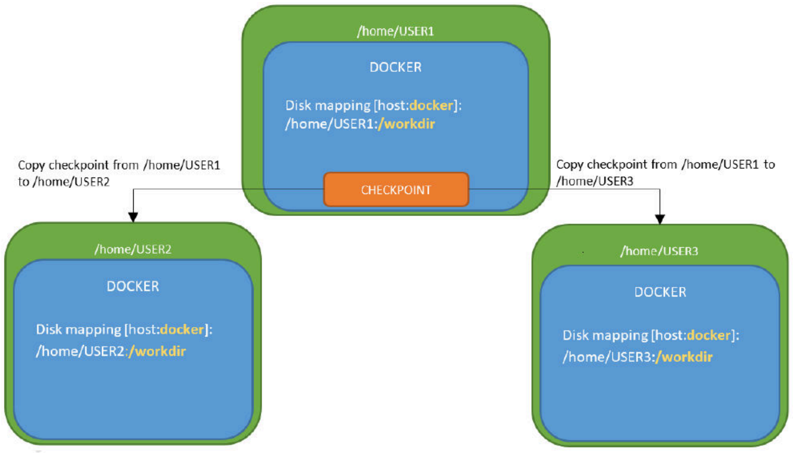
Docker rootless can be installed only on CentOS 7.3 host internally. Refer to the following for the description: https://synopsys-assist.onbmc.com/dwp/app/#/knowledge/KBA00010622/rkm

What Docker image consists of?

Image consists of files, libraries and tools installed during the build phase. The commands to install packages and to configure are available in the Dockerfile.

How is path-translation/virtualization happening when using containers?

Consider the scenario from the following image:


/home/USER1
DOCKER
Disk mapping [host:docker]:
/home/USER1:/workdir
CHECKPOINT
Copy checkpoint from /home/USER1 to /home/USER2
Copy checkpoint from /home/USER1 to /home/USER3
/home/USER2
DOCKER
Disk mapping [host:docker]:
/home/USER2:/workdir
/home/USER3
DOCKER
Disk mapping [host:docker]:
/home/USER3:/workdir

User1 has a project in `/home/USER1 (SNPS_SHARED_FOLDER=/home/USER1)`, and mounts it to Docker. Thus, in the container it is the `/workdir (DOCKER_MOUNT_FOLDER=/workdir)`. User1 creates a checkpoint in a standard way (refer to Performing a Checkpoint) in the `/workdir` directory inside the container, which in fact is in `/home/USER1`.

Now User2 and User3 are copying this project to their `/home/USER2` and `/home/USER3` (setting `SNPS_SHARED_FOLDER` accordingly). They start Docker while mapping their directories to `/workdir` (using `DOCKER_MOUNT_FOLDER=/workdir`).

Therefore, the checkpoint is physically located in `/home/USER2` and `/home/USER3`, but in their containers it is in `/workdir` similar to USER1. Hence, from the checkpoint perspective, it is in the same location. Restart is done in a standard way by calling `dmtcp_restart_script_zebu.sh`.

How is this path virtualization/translation happening comparing to pathvirt?

Pathvirt does path translation on the go during restart, based on the environment variables. Docker does not perform path translation on the go, since, after the container is started, and disks/folders are mapped paths are the same inside the container.

Dockerfile for Docker is basing on different OS than installed on host. How does it work when there are two OSes?

Docker do not have OS. The image contains all necessary libraries/executable from the base OS that you have selected in the Docker file that can run inside the container. Inside the container, you still have the same kernel as on the Host OS.

What if User1 saves checkpoint without container and User2 restarts it in Docker container?

All the users sharing checkpoints are using Docker, so User1 is also using Docker.

User1 is sharing checkpoint with five users which are restarting it, can the users use the same container?

Each user is copying the User1 checkpoint to their location and each user starts the container. The container can be based on the same image, so it should be possible to build the image once for general usage. It may require additional permissions as the image files are stored on no- NFS disk (usually `/SCRATCH/<username>`).

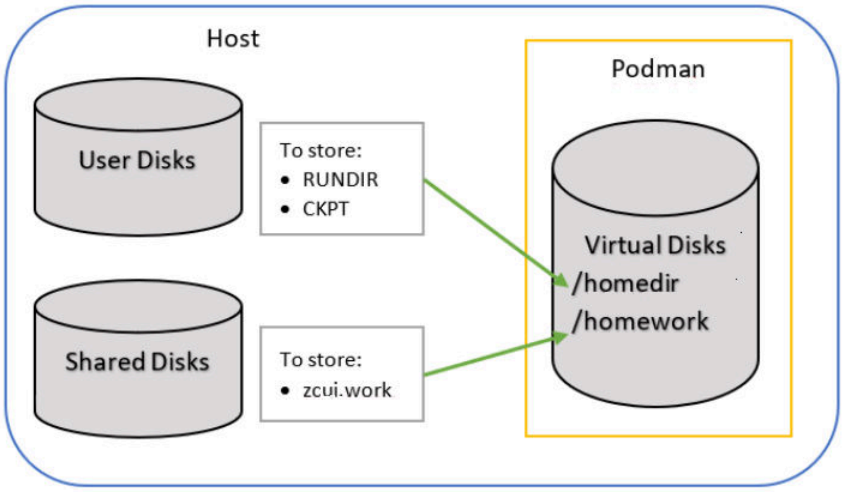
What is the life cycle of Image/Container?

Container is the process that uses image to execute things. Container life cycle starts when user starts it using `run_docker.sh` and ends when the user exits. The image resides on the non-NFS disk, typically under `/SCRATCH/<username>`. The life cycle ends when a user deletes files from this directory.

Podman Container Support

Podman is a daemon-less container engine for developing, managing, and running OCI (Open Container Initiative) containers on Linux systems. Users must mount proper directories and configure necessary environment settings to use ZeBu with DMTCP in a Podman container. This section describes how to use Podman in Synopsys environment to run ZeBu with DMTCP in a hermetic Linux OS container.

Figure 5 Mapping host disks into Virtual disk



Podman Container Build

Podman helper scripts are available at `$ZEBU_ROOT/thirdparty/dmtcp/scripts/podman/`.

Copy the scripts to a folder, set the `SNPS_PODMAN_IMAGE_NAME` variable with a name of the image and execute the following commands:

```
chmod 777 ./build_podman.sh
export SNPS_PODMAN_IMAGE_NAME=<YOUR_IMAGE_NAME>
./build_podman.sh
```

Podman Container Run

Before running the Podman, source environmental settings from ZeBu and set the `ZEBU_SYSTEM_DIR` environment variable to correspond to the ZeBu configuration which you are going to use. In addition, configure the following variables:

- `SNPS_PODMAN_IMAGE_NAME` – name of the image.
- `SNPS_SHARED_FOLDER` – a path to folder to share with Podman.
- `PODMAN_MOUNT_FOLDER` – a path to folder in Podman which will be used to mount `SNPS_SHARED_FOLDER`.
- `XDG_RUNTIME_DIR` – path to non NFS directory (same as `graphroot` from `~/.config/containers/storage.conf`).

```
chmod 777 ./run_podman.sh
export SNPS_PODMAN_IMAGE_NAME=<YOUR_IMAGE_NAME>
export SNPS_SHARED_FOLDER=<YOUR_FOLDER>
export PODMAN_MOUNT_FOLDER=<FOLDER_IN_PODMAN>
export XDG_RUNTIME_DIR=/SCRATCH/<username>
./run_podman.sh
```

Note:

`YOUR_FOLDER` should point to a folder you want to mount in the Podman Linux system. In case of any errors on mounting your folder in the Podman, refer to Podman documentation.

When sharing checkpoints between different hosts using Podman, make sure the hosts have identical Linux OS.

Running Design Folders in Podman

To mount more design folders in Podman, follow these instructions:

1. Open run_podman.sh script and find the following lines in the script:

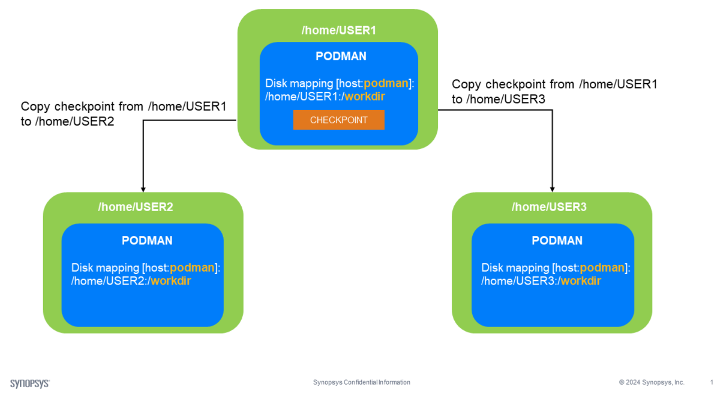
```
#array of user paths mapped to Podman
# format is ( ["user_path"]="path_in_podman"
["second_user_path"]="path_in_podman")
declare -A user_paths=([
"$SNPS_SHARED_FOLDER"]="$PODMAN_MOUNT_FOLDER")
```

2. After declare, enter the following line to add new folders to be mount:

```
user_paths+=( ["<NEW_DESIGN_FOLDER>"]="<NEW_FOLDER_IN_PODMAN>" )
```

Path-Translation/Virtualization Using Containers

Consider the scenario discussed in the following image.


/home/USER1
PODMAN
Disk mapping [host:podman]: /home/USER1:/workdir
CHECKPOINT
Copy checkpoint from /home/USER1 to /home/USER2
Copy checkpoint from /home/USER1 to /home/USER3
/home/USER2
PODMAN
Disk mapping [host:podman]: /home/USER2:/workdir
/home/USER3
PODMAN
Disk mapping [host:podman]: /home/USER3:/workdir
synopsys
Synopsys Confidential Information
© 2024 Synopsys, Inc.
1

User1 has a project in `/home/USER1 (SNPS_SHARED_FOLDER=/home/USER1)`, and mounts it to Podman. Thus, in the container it is the `/workdir (PODMAN_MOUNT_FOLDER=/workdir)`. User1 creates a checkpoint in a standard way (refer to Performing a Checkpoint) in the `/workdir` directory inside the container, which in fact is in `/home/USER1`.

Now User2 and User3 are copying this project to their `/home/USER2` and `/home/USER3` (setting `SNPS_SHARED_FOLDER` accordingly). They start Podman while mapping their directories to `/workdir` (using `PODMAN_MOUNT_FOLDER=/workdir`).

Therefore, the checkpoint is physically located in `/home/USER2` and `/home/USER3`, but in their containers it is in `/workdir` similar to USER1. Hence, from the checkpoint perspective, it is in the same location. Restart is done in a standard way by calling `dmtcp_restart_script_zebu.sh`.

Feedback

12

Appendix A: Basic Checkpoint Sharing Use Model Example

This appendix discusses the following topics:

- Use Models

Use Models

This section discusses the following two use models:

- Use Model 1: User1 saves the checkpoint and also restores it on same/different host.
- Use Model 2: User1 saves the checkpoint but shares it with user 2 who in turn restores the checkpoint on same/different host.

  Note:

  DMTCP package must be installed on every runtime host you plan to run DMTCP on unless the hosts have same configurations.

  If you have a UART transactor, then run it in the Server mode only. You can use the bridge solution if you want “xterm” to pop-up despite running server mode.

  You will need the 7zip compression utility.

Step 1: Installing DMTCP

To install DMTCP:

1. Copy the latest package to your runtime area from: `$ZEBU_ROOT/thirdparty/dmtcp/dmtcp*.tar.gz`.

   For example, `$ZEBU_ROOT/thirdparty/dmtcp/dmtcp-2.6.0-snps16.1.tar.gz`

   Sometimes the package in `$ZEBU_ROOT` area can be older. Make sure you take a new package from: `/remote/vginterfaces1/dmtcp/packages/` area
2. Login to the host.

3. Source your Zebu setup. Make sure `$ZEBU_ROOT` is set and you are using 2021.09-1.B4 or higher.
4. Set the GCC version: gcc 9.2.
5. Untar the package.

   ```
   $ZEBU_ROOT/thirdparty/dmtcp/scripts/install_dmtcp.sh -i dmtcp-2.6.0-
   snps17.7.tar.gz -f -d -b -o <full path to the area you want to install
   dmtcp>
   ```

   For example: $`ZEBU_ROOT/thirdparty/dmtcp/scripts/install_dmtcp.sh -i dmtcp-2.6.0-snps17.7.tar.gz -f -d -b -o /remote/vginterfaces1/dmtcp/dmtcp-2.6.0-snps17.7/`

Note:

if you have a 32-bit binary (for example: `cm7_decoder`), then you will need a package that is 32-bit compliant, it is not officially recommended. In such cases, it is recommended to install a separate package with an “-m” option and mention in the naming of the package as 32-bit.

Step 2: Environment Setup

These environment variables are required only at runtime.

Set the following environment variables:

```
setenv DMTCP_DL_PLUGIN 0
setenv ZEBU_INTERNAL_SIGNAL PWR
setenv ZEBU_DONT_CANCEL_THREADS OK
setenv ZEBU_DMTCP_ENABLE_MULTI_PROCESS true
setenv DMTCP_PATH $ZEBU_ROOT/thirdparty/dmtcp/dmtcp-2.6.0-snps16.1.tar.gz
setenv DMTCP_HOME "${DMTCP_PATH}"
setenv PATH ${DMTCP_PATH}/bin:${DMTCP_PATH}/plugin/batch-queue:${PATH}
setenv LD_LIBRARY_PATH ${DMTCP_PATH}/lib/dmtcp:${LD_LIBRARY_PATH}
setenv PATH $ZEBU_ROOT/thirdparty/dmtcp/7Zip:$PATH
setenv DMTCP_FORKED_CHECKPOINT 1
```

Step 3: zRci

```
dmtcp_launch --with-signal-opt --new-coordinator --modify-env --7z --7zmx
 2 zRci run.tcl |& tee -i run.log
```

Step 4: Checkpoint Save

After running for some clock cycles, you may want to save the checkpoint.

If you have a socket connection open via Net Connect (nc), then make sure you close/disconnect it. Otherwise, this procedure fails and your system will stop responding. Instead of Net Connect for UART, you can also explore the option of **zCheckpointingBridge** solution.

Note:

Before checkpoint save, make sure you have closed all wavedumps, disabled `ccalls` and stopped the clocks.

For use case 2, you need to ensure that you have closed all monitor/debug logs from transactor.

On the **zRci** prompt, enter: `checkpoint -fullsave at1`.

Where, `at1` indicates the name of the your checkpoint directory.

In this example, the runtime directory as mentioned in **zRci** is `ZRCI_1`. This creates a folder in your runtime directory (`ZRCI_1`): at1.

Depending on the size of your design and the number of ZRMs, this step takes some time to complete. You can exit from **zRci** after the checkpoint is saved.

Note:

- Since the environment variable `"DMTCP_FORKED"CHECKPOINT"` is declared, the **zRci** shell is returned while the checkpoint save continues in the background.
- Do not quit **zRci** till that point.
- Notice the file `ckpt_*.dmtcp.temp` in the checkpoint directory “at1”.
- After the checkpoint is fully saved in the background, the temp file is completed.
- The following message indicates that the background save is completed: `--ZeBu : zRci : Moving the checkpoint directory "./ZRCI_1/at1"` done by script pid.
- Exit from **zRci** after the save is completed.

Step 5: Fast State Conversion

Note:

This step does not need emulator and it should be performed only once per checkpoint.

Syntax

```
$ZEBU_ROOT/thirdparty/dmtcp/scripts/convert_ckpt_hw_state.sh -z
<zebu.work> -c <checkpoint directory>
```

For example: `$ZEBU_ROOT/thirdparty/dmtcp/scripts/convert_ckpt_hw_state.sh -z zebu.work -c ZRCI_1/at1/`

Use Model 1

Perform the following additional steps for use model 1, in continuation with the command steps.

- Step 6: Checkpoint Restore

Step 6: Checkpoint Restore

You can either restore the checkpoint from same terminal, or from a new terminal; either using the same host, or a different host.

To restore the checkpoint:

1. Take a new terminal and login to the host.
2. Set the following environment variables:
   - `ZEBU_PHYSICAL_LOCATION`
   - `setenv PATH $ZEBU_ROOT/thirdparty/dmtcp/7Zip:$PATH`
   - Add `gcc 9.2` to your `$PATH`
3. Run the following command:

```
./<runtimedirectory>/<checkpoint
 directory>/dmtcp_restart_script_zebu.sh ORIG_DISPLAY=$DISPLAY |& tee
 -i restore.log
```

For example: `./ZRCI_1/at1/dmtcp_restart_script_zebu.sh ORIG_DISPLAY= $DISPLAY |& tee -i restore.log`

Note:

Always restore from your runtime directory. Otherwise, the restore process errors out. This step takes some time, but it restores at the point you had initially saved for.

Use Model 2

Perform the following additional steps for use model 1, in continuation with the command steps.

- Step 7: File Permissions
- Step 8: Shadow Checkpoint
- Step 9: Checkpoint Restore

Note:

The runtime directory is assumed as `zebu_run`.

In this use case, User1 will share the entire runtime directory with User2 so that User2 can access the checkpoint saved by User1.

Step 7: File Permissions

User1 must make sure that the `zrci_command.log` has permissions set to 777.

If any monitor/transactor log was present but was not closed while saving the checkpoint, then set the file permission of these log files to 777.

Step 8: Shadow Checkpoint

Syntax

```
$ZEBU_ROOT/thirdparty/dmtcp/scripts/checkpoint_shadow.sh
-n <user2runfolder> -r <user1runarea> -c <runtimezrcidb>/
<checkpointtobecloned> -a
```

For example:

```
$ZEBU_ROOT/thirdparty/dmtcp/scripts/checkpoint_shadow.sh -n myruntimedir
-r /project/user1/testProject -c ZRCI_1/at1 -a
```

Step 9: Checkpoint Restore

User 2 must perform the following steps to restore the checkpoint:

1. Login to the host.
2. Navigate to the cloned area folder.

   For example: `myruntimedir`

3. Set the following environment variables:
   - `ZEBU_PHYSICAL_LOCATION`
   - `setenv PATH $ZEBU_ROOT/thirdparty/dmtcp/7Zip:$PATH`
   - Add `gcc 9.2` to `$PATH`.

     For example: `./ZRCI_1/at1/dmtcp_restart_script_zebu.sh ORIG_DISPLAY= $DISPLAY USER=$USER HOME=$HOME LOGNAME=$USER XAUTHORITY=$XAUTHORITY |& tee -i restore.log`

Note:

Always restore from your runtime directory. Otherwise, the restore process will error out. This step will take some time, but it will restore at the point you had initially saved for.

13

Appendix B: Using GDB with DMTCP

This appendix discusses the following topics:

- Debug on Restart

Debug on Restart

The GNU Project Debugger (GDB) allows debugging programs while they are running.

Although, in some cases, it was observed that GDB is unable to get all necessary details from the application to display the correct information while debugging DMTCP restart. It appeared as though debug symbols were not loaded correctly from the application.

The `save-symbol-files-to-gdb-script.py` script resolves this issue.

```
python <path_to_script>/save-symbol-files-to-gdb-script.py <PID>
 > ./<PID>_gdb_symbols
```

where,

`<path_to_script>` indicates the location: `<DMTCP_package>/util/`.

Perform the following steps to use this script and debug DMTCP during restart:

1. Enable DMTCP debug on restart: `setenv DMTCP_RESTART_PAUSE 1`.
2. Restart by calling `<Path to ckpt>/dmtcp_restart_script_zebu.sh`.
3. Check the log file and search for GDB. Alternatively, use top Linux command and search for `mtcp_restart`.

   Note:

   If more then one processes are running in DMTCP, it is necessary to connect GDB to all the processes.

   ```
   grep -A 5 -B 2 "gdb" <path_to_log_file>
   ```

Example 1 Example Log

The following information is available in the log from DMTCP:

```
<process name> mtcp_restart.c:1710 remapMtcpRestartToReservedArea:
          For debugging (<process name>):
            (gdb) add-symbol-file mtcp_restart 0x2aaaaed76020
        [<process pid>] mtcp_restart.c:550 restart_fast_path:
          We have copied mtcp_restart to higher address.  We will now
            jump into a copy of restorememoryareas().
        [<process pid>] mtcp_restart.c:685 restorememoryareas:
          Entering copy of restorememoryareas().  Will now unmap old
 memory
        --
        Attach to the computation with GDB from another window:
        (This won't work well unless you configure DMTCP with
 --enable-debug)
          gdb PROGRAM_NAME(<process name>) <process pid>
        You should now be in 'ThreadList::postRestartDebug()'
          (gdb) list
  (gdb) p dummy = 0
```

If you do not see this information, then use top to get information about PID numbers.

4. Generate symbols for all processes:

```
python <path_to_script>/save-symbol-files-to-gdb-script.py
 <process_PID> > ./<process_PID>_gdb_symbols
```

5. Start GDB (one per process).
6. Source generated symbol per process: `source ./<process_PID>_gdb_symbols`.

   Note:

   Repeat this step for every GDB/process.
7. Connect to the processes: `attach <process_PID>`.

   Note:
   Repeat this step for every GDB/process.
8. Do not update the symbols from `mtcp_restart`.

   Note:

   If GDB prompts you to update the debug symbols from `mtcp_restart` application, select `no`.
9. In GDB, display the code and observe that the process is in `while(dummy);` loop.
10. Change `dummy` to `0` to debug both processes.

```
p dummy = 0;
```

Note:

Repeat this step for every GDB/process.