Towards Non-Intrusive SystemC Checkpointing for Digital Virtual Prototypes

Deepak Ravibabu[1], Muhammad Hassan[1, 3], Thilo Vörtler[2], Karsten Einwich[2], Rolf Drechsler[1, 3], and Daniel Große[1, 4]

[1]Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany
Email: {deepak.ravibabu, muhammad.hassan}@dfki.de
[2]COSEDA Technologies GmbH, 01099 Dresden, Germany
Email: {thilo.voertler,karsten.einwich}@coseda-tech.com
[3]Institute of Computer Science, Bremen University, 28359 Bremen, Germany
Email: drechsler@informatik.uni-bremen.de
[4]Institute for Complex Systems, Johannes Kepler University, 4040 Linz, Austria
Email: daniel.grosse@jku.at

Abstract

Checkpointing enables the storage and restoration of the simulation state of *Virtual Prototype*s (VPs), significantly reducing the debugging and testing cycle times, thereby accelerating the overall development process. In this work, we present a novel methodology for checkpointing digital SystemC VPs, with a particular focus on storing and restoring SC_THREAD processes, which are integral to SystemC models. The proposed checkpointing methodology is non-intrusive to the SystemC kernel and is implemented as a SystemC library, which integrates seamlessly with existing digital VPs with minimal effort. The effectiveness of the proposed methodology is demonstrated through a case study on a digital *Finite Impulse Response* (FIR) filter. The filter's state was successfully restored from a checkpoint, and its execution was validated to be consistent with the filter's expected behavior. The results confirm that the proposed checkpointing library reliably restores the simulation state of digital VPs, enabling faster design iterations.

1 Introduction

The automotive industry is undergoing a significant transformation, driven by the increasing complexity of systems from advancements in autonomous driving and electrification. As modern vehicles become more reliant on a sophisticated interplay between hardware and software components, automotive manufacturers face the challenge of integrating vast amounts of sensor data and managing high computing demands. Addressing these complexities requires a holistic approach that models the interactions between all system components, rather than analyzing subsystems like *Electronic Control Unit*s (ECUs) in isolation [1].

*Virtual Prototype*s (VPs) [2] have emerged as a key technology in addressing these challenges. VPs enable the simulation of complex automotive systems prior to the availability of physical hardware, offering a cost-effective and efficient means to develop, test, and validate system designs. They are particularly valuable during early-stage software development, as they facilitate the integration of hardware and software components at multiple abstraction levels. The software executed on a VP is identical to that on real hardware, ensuring consistency across the development process. The adoption of the shift-left methodology, which emphasizes early testing and validation during the development life cycle, has further reinforced the value of VPs in automotive design. This approach minimizes costs and ensures correctness by identifying potential issues at the earliest possible stage. SystemC [3], a system description language, is heavily used in industrial practice to serve as a foundation for modeling VPs. Its C++-based class library supports the efficient simulation of heterogeneous components, providing the flexibility required to meet the diverse demands of modern automotive electronics [4, 5].

Despite their advantages, VPs face challenges in managing intricate simulation scenarios. Some of the key bottlenecks include prolonged simulation times for complex tasks such as device initialization to specific internal states (e.g., booting an *Operating System* (OS)), failure recovery, and running extended test scenarios. Addressing these challenges is crucial to maximizing the potential of VPs in modern automotive development.

Checkpointing [6] offers a promising solution to these challenges. This technique involves storing and restoring the state of a simulated system, enabling simulation to resume from a specific point. This capability is especially valuable for developers needing to revert to a checkpointed state prior to an error or fast-forward to a specific point in time without executing the entire simulation from the beginning.

In this paper, we present a methodology for checkpointing SystemC digital VPs, focusing on SC_THREAD processes, given their extensive use in VPs to simulate

concurrent behavior. Our methodology leverages the Accellera SystemC kernel [7] and the QuickThreads [8] package, to store and restore the execution state of a system. It enables efficient checkpointing without requiring any modifications to the SystemC kernel. The proposed checkpointing methodology is implemented as a SystemC library and its simulation run time benefit is demonstrated using a SystemC VP that implements a *Finite Impulse Response* (FIR) filter, provided by our industrial partner.

The paper is structured as follows. It discusses briefly existing works related to checkpointing SystemC VPs in Section 2. Then, a short description of the preliminaries along with a running example is given in Section 3. Then, an overview of the proposed methodology is presented in Section 4. Later the implementation of the checkpointing methodology along with its working on the running example is explained in Section 5. A case study to show the applicability of the proposed methodology is given in Section 6. Finally, the limitations of our methodology and the future work are presented in Section 7, and the paper is concluded in Section 8.

2 Related Works

The concept of checkpointing initially involved storing the entire system state, including the state of the OS with all its running applications. With the advent of *Virtual Machine* (VM) [9], system-level checkpointing is made possible. In the checkpointing solution provided in [10], the entire state of a system, including its heap memory, stack memory, and OS resources, is checkpointed. To enable checkpointing of OS resources, the framework relies on a kernel module. This module integrates with the Linux kernel to capture and restore low-level system details that are inaccessible from user space. While this framework itself does not inherently support SystemC VPs, with modifications or patches to the SystemC kernel it might be possible to enable checkpointing.

In contrast, the work presented in [11] introduces a user level checkpointing library, *Libckpt*, designed to provide portable checkpointing without requiring kernel modifications. Developers can link their applications with the library to enable checkpointing functionality without needing any custom implementation. Some of its useful features are smaller checkpoint size, incremental checkpointing and forked checkpointing (simulation executed parallel to checkpointing). The library is highly efficient for single-process scenarios, but its primary limitation is the lack of support for checkpointing parallel processes.

The previous checkpointing techniques were applied on VM level, later such techniques were adopted to the domain of SystemC VPs. The work in [12], based on the Virtutech Simics [13] simulator, introduces a technique for checkpointing SystemC VPs by manually marking the system states to be checkpointed. In this technique, only the essential information which constitutes the system state is stored and restored by serialization and deserialization. This work targets SystemC SC_METHOD processes but excludes restoring stack-based state of SC_THREAD processes.

A checkpointing framework for SystemC simulations was proposed in [14] for automating the handling of most SystemC VPs. Even though host OS resources are not checkpointed, this method provides hooks for developers to manage them. It enables periodic checkpointing, simulation state transfer, and debugging workflows with minimal code modifications. In the extended work [15], they build to provide support for checkpointing external applications, such as debuggers and graphical interfaces, re-establishing connections seamlessly after a restore, but the gap in handling SC_THREAD processes remains unaddressed.

In the thesis work [16], a standardized framework is developed for snapshotting SystemC *Transaction Level Modeling* (TLM) based VPs, ensuring compatibility with SystemC standards and tools. The framework introduces lightweight and portable snapshots, enabling seamless restoration across platforms and updated models. A snapshot manager class is introduced to handle simulation state serialization and restoration, automating the checkpointing process for most SystemC VPs. The framework supports integration with the *Universal Verification Methodology* (UVM) [17], enhancing test automation and CI workflows. The case studies demonstrate reduced testing time and overhead, improving productivity in pre-silicon validation workflows. In the work [18], time decoupling method is used to roll back the simulation with the usage of UNIX fork() system call, which result in performance overhead. While the approach is innovative, it is less portable and difficult for debugging purposes.

Existing checkpointing solutions have demonstrated effective mechanisms for saving and restoring simulation states in certain contexts, showcasing their utility in improving simulation efficiency and debugging workflows. However, these existing checkpointing solutions fail to support SC_THREAD processes, essential for modeling stateful and event-driven behaviors, making them integral to many VPs. In this work, we addresses this key limitation, enabling state restoration for SystemC-based digital VPs.

3 Preliminaries

This section provides relevant background and a motivating example to explain the proposed methodology.

3.1 SystemC

SystemC is a widely adopted system-level modeling language that facilitates the simulation and design of hardware and software components. A SystemC VP is composed of interconnected modules, each representing various parts of the system. These modules encapsulate

```
1  SC_MODULE(counter_mod){
2    sc_out<double> outp; //port declaration
3    void count_process(); // thread process
4    SC_THREAD(count_process);
5    private:
6    sc_time wait_time = 1_SC_US;
7    void trigger(unsigned long int cnt1);
8    int target_cnt = 20;
9  };
10
11 void counter_mod::count_process()
12 {
13   unsigned long int cnt1 = 0;
14   while (cnt1 < target_cnt) {
15     sc_core::wait(wait_time);
16     cnt1++;
17     std::cout << "cnt1: " << cnt1 << std::endl;
18   }
19   trigger(cnt1); // calls trigger function after
         reaching target count value
20   outp.write(cnt1);
21 }
22
23 void counter_mod::trigger(unsigned long int cnt1){
24   // generate the trigger
25 }
```

Listing 1 SystemC Counter Module

ports for communication, processes for functionality, and internal variables to store system state, enabling a modular and hierarchical design approach. In SystemC modules, parameters and processes play distinct roles in defining its structure and behavior. **Parameters** govern the module's configurable properties, such as internal variables and constants. **Processes** define the functional logic of a module, including computations and interactions with other modules, while coordinating with the SystemC kernel for event-driven execution and simulation scheduling.

Processes in SystemC are sensitive to specific events and are triggered accordingly. The two primary types of processes are the SC_METHOD and SC_THREAD. SC_METHOD processes are event-driven and execute fully once triggered, without the ability to suspend during execution. These processes are placed back in the queue until their triggering event occurs again. SC_THREAD process can be suspended and resumed during execution. This is typically achieved using *wait()* statements, making them suitable for complex behaviors that require control over execution flow. To schedule the module processes during a simulation run, the SystemC kernel utilizes a scheduler queue.

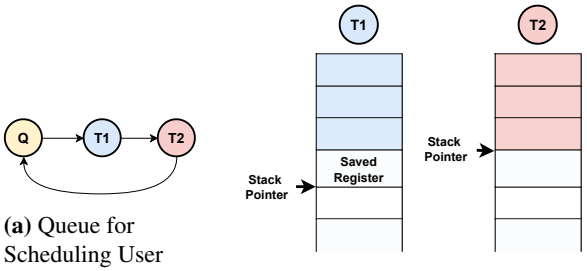
3.2 Running Example

To illustrate the proposed checkpointing methodology, we present a basic SystemC module that implements a counter, as shown in Listing 1.

The counter module declares its ports, parameters and processes between Line 1 and Line 10. The module parameters include the variables *target_cnt*, which specifies the target count value, and *wait_time*, which represents the time interval for incrementing the counter. The module process *count_process()* represents the core functionality of the module. The implementation detail of *count_process()* spans Line 11 to Line 21. The process initializes a local variable *cnt1* and increments it at an interval of *wait_time* (set to 1 µs in this example). The process continues until *cnt1* reaches the specified *target_cnt* (set to 20 in this example). Upon reaching the target count, the process invokes the *trigger()* function and writes the final count value to the output port *outp*.

3.3 QuickThreads

QuickThreads [8], a user thread package used by the SystemC kernel, manages thread initialization and context-switching, facilitating transfer of control between threads during execution. During a context switch, it enables thread context to be saved, allowing the thread to later resume execution from its saved state when invoked. The proposed methodology leverages the QuickThreads library for accessing the thread context of each instantiated SC_THREAD module processes, enabling checkpointing. The application of QuickThreads package and a comparison with other available thread packages is presented in [19]. An example of a context switch with two user threads and a scheduler queue is shown in Figure 1.



(a) Queue for Scheduling User Thread

(b) Stack Memory of User Threads

Figure 1 Stack Layout during Thread Context Switching
Q: Scheduler Queue
T1: User Thread1
T2: User Thread2

The threads are tracked using the scheduler queue *Q* (depicted in Figure 1a), maintained by the simulation kernel. The user thread *T1* is first scheduled for execution followed by *T2*. Each user thread is allocated a unique stack memory during its execution.

An example for a context switch scenario, where thread *T1* is blocked and the next thread *T2* is executed is illustrated in Figure 1b (the stack grows down). The context switch is performed by QuickThreads before the execution of *T2*.

First, along with the existing local variables of the old thread *T1*, the current host register values (representing its current simulation state) are stored in its stack, adjusting the stack pointer accordingly. A *helper* function is then invoked to update the scheduler queue, placing the blocked thread *T1* back with its updated stack pointer. Next, the stack pointer is switched to the stack of the new thread *T2* and old thread *T1* is suspended. Once the execution of *T2* is complete, *T1* is scheduled to resume execution. At this point, the *helper* function retrieves the stored host register values from the stack of *T1* and restores its execution state, allowing *T1* to continue from where it was suspended.

```
1  SC_MODULE(counter_mod){
2    sc_out<double> outp; //port declaration
3    void count_process(); // thread process
4    SC_THREAD(count_process);
5    private:
6    sc_time wait_time = 1_SC_US;
7    void trigger(unsigned long int cnt1);
8    int target_cnt = 20;
9    // marked for checkpointing
10   cos_sc_variable_handle<int> target_cnt_vh = {
         target_cnt, "target_cnt_handle"};
11 };
12
13 void counter_mod::count_process()
14 {
15   unsigned long int cnt1 = 0;
16   while (cnt1 < target_cnt) {
17     sc_core::wait(wait_time);
18     cnt1++;
19     std::cout << "cnt1: " << cnt1 << std::endl;
20   }
21   trigger(cnt1); // calls trigger function after
         reaching target count value
22   outp.write(cnt1);
23 }
24
25 void counter_mod::trigger(unsigned long int cnt1){
26   // generate the trigger
27 }
```

Listing 2 SystemC Counter Module - Checkpointed

4 Proposed Methodology

In this section, the method to enable checkpointing of the running example and the execution flow of the proposed checkpointing methodology are discussed.

4.1 Concept

The proposed methodology focuses on checkpointing two key aspects which determine the functionality of a module: *(i)* the values of module parameters, and *(ii)* the simulation state of module processes. To checkpoint the module parameters, developers should explicitly mark them by defining corresponding objects using a wrapper class. In contrast, checkpointing the simulation state of module processes requires no additional user module modifications, as the proposed checkpointing library handles this automatically.

For demonstration, we apply the proposed methodology to the running example (Listing 1). This enables the simulation to save its state at a checkpoint and later resume execution from the saved point. By eliminating the need to re-execute computations performed before the checkpoint, it significantly improves efficiency in simulation workflows. The checkpointed version of the running example is shown in Listing 2.

To track module parameters, the proposed library uses a wrapper class (*cos_sc_variable_handle*, available in COSIDE [20]). In the example, the module parameter *target_cnt* is tracked by defining an object handle *target_cnt_vh*, as shown in Line 10 in Listing 2.
To checkpoint the simulation state of the process (*count_process*), there is no additional modification required to the user module. When the checkpoint condition is met, the library automatically stores the simulation state of the thread which is accessible using QuickThreads.

4.2 Overview of the Execution Flow

The proposed checkpointing methodology for storing and restoring the counter module parameters is outlined as follows. At the start of the simulation, the modules are instantiated, along with their module parameters and the corresponding object handles that track these parameters.
During simulation execution, when the checkpoint condition is met, the checkpointing library iterates through all instantiated objects to identify the object handles tracking the module parameters. The methods defined within this wrapper class are then invoked to save the values of the tracked module parameters to a checkpoint file.
Similarly, during execution in restore mode, the checkpointing library identifies the object handles referencing the module parameters. Using the internal methods of the wrapper class, the module parameter values are restored to match the values saved in the checkpoint file, ensuring consistency with the checkpointed state.

The overview of the proposed checkpointing methodology for storing and restoring the counter module process is depicted in Figure 2.

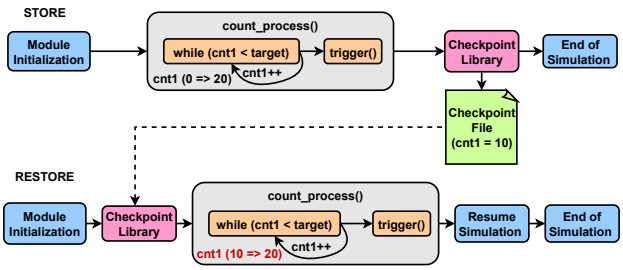


Figure 2 Visualizing Checkpointing of count_process() from Listing 2

The top section of Figure 2 illustrates the workflow for checkpointing the module process (*count_process()*, depicted in gray). The counter module is compiled with the checkpoint library and executed in store mode, with the checkpoint set at a simulation time of 10 µs. The *count_process()* thread begins incrementing the variable *cnt1* at intervals of 1 µs. When the simulation time reaches the specified checkpoint time (10 µs), the checkpoint library (depicted in pink) is invoked to store the simulation state of the process. The checkpoint condition is checked at the *sc_core::wait()* statement, where the simulation kernel is involved. At this point, the simulation state of the process, including the value of *cnt1* (*cnt1 = 10*) present in the thread stack region, is stored in a local checkpoint file (depicted in green).

The workflow for restoring the module process (*count_process()*) is depicted in the bottom section of Figure 2. At the beginning of the simulation run in

restore mode afterwards, the modules are instantiated, and the local variable *cnt1* inside the process is instantiated to 0 since execution has not yet begun. The checkpointing library (depicted in pink) is then invoked before process execution, to restore the simulation state of the process. The checkpoint file (depicted in green) is analyzed, and the saved simulation state data is used to update the newly instantiated *count_process* thread. After restoring, the simulation resumes, and *count_process* executes with *cnt1 = 10* rather than starting from 0 (depicted in red).

5 Implementation of Checkpointing Methodology

In this section the implementation details of the proposed checkpointing methodology applied to the illustrative example are explained in detail.

5.1 SystemC Module Parameters

SystemC modules are instantiated dynamically, with their module parameters allocated in heap memory (a dynamically managed memory region used for runtime object allocation). The memory addresses of these parameters typically remain unchanged during their lifetime, facilitating efficient memory management and ensuring consistent access during the simulation.

The custom wrapper class (*cos_sc_variable_handle*) available in the COSIDE *Electronic Design Automation* (EDA) tool allows efficient tracking, serialization, and deserialization of the variable being referenced. It does not directly enable checkpointing but, the wrapper class is leveraged in the checkpointing library to implement the store/restore functionality for module parameters. In our prototype implementation, the wrapper class provides the following key methods:

Store: The *serialize()* method converts the value of a tracked parameter into a string representation using *std::ostringstream*. This serialized data is saved to a checkpoint file.
Restore: The *deserialize()* method parses the saved string representation from the checkpoint file and updates the parameter value accordingly, thereby restoring the module parameter's value with the same value in the stored checkpoint.

For storage of module parameters in the checkpoint file, a hierarchical naming system (*topmodule.submodule.variable*) is used, making it easier to identify and access them. During restoration, the instantiated object handle for the module parameter in the current simulation is matched with the hierarchical name in the checkpoint file. If they are identical, the value of the module parameter is updated with the value stored in the checkpoint file.

5.2 SystemC Module Process

The simulation state of an SC_THREAD module process, or *thread context*, consists of values stored in the thread's stack memory and the host processor registers. Checkpointing an SC_THREAD process involves capturing this thread context and carefully restoring it. The process for checkpointing the SC_THREAD process is illustrated in Figure 3, which outlines both store and restore modes.

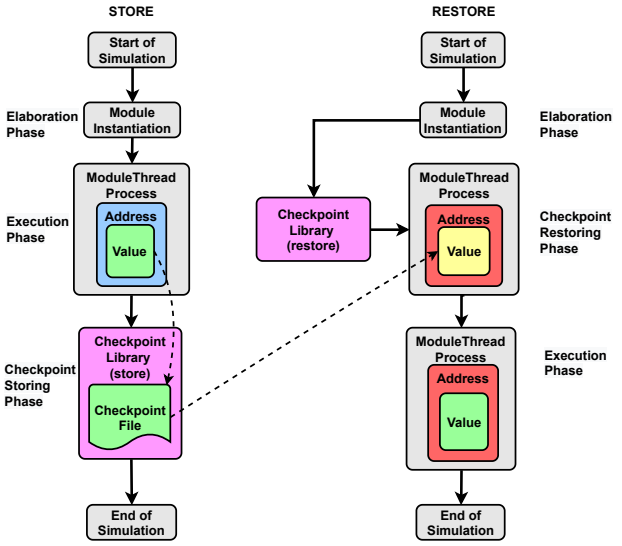


Figure 3 Checkpointing SystemC Thread Process

Checkpointing - Store: The left section of the Figure 3 depicts the execution flow of a SystemC module compiled with the checkpoint library and executed in store mode. During the *elaboration phase*, all modules are instantiated and their thread processes are instantiated, with each thread allocated a dedicated stack memory region (stack addresses shown in blue, values in green). When a thread process is suspended at a *wait()* statement, the SystemC kernel switches execution based on the scheduler queue. If the current simulation time matches the given checkpoint time (default: end of simulation), control is passed to the checkpointing library (depicted in pink). During the context switch, the checkpoint library accesses the thread's simulation state (stack memory region) using QuickThreads and stores it onto the checkpoint file (depicted in green). Then, the SystemC kernel resumes control and the simulation concludes.

Checkpointing - File Analysis: From the execution in store mode, the checkpoint file containing the stored thread context (stack memory region) is analyzed. Consider a thread process with a main function *func1*, which makes a nested function call to *func2*. When the checkpoint condition is reached, if the thread process was executing *func2*, then its thread context layout at that point is illustrated in Figure 4. It is important to note that the stored thread context includes both the stack memory addresses and their corresponding values, which are critical for analysis.

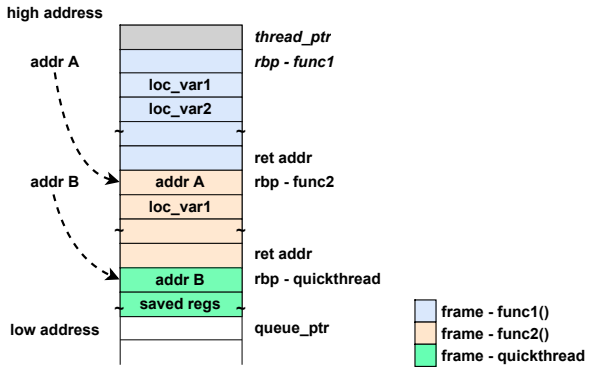


Figure 4 Layout of Stored Thread Context

The stored thread stack at the *wait()* statement is observed to follow a consistent pattern, enabling the identification of individual stack frames for each function call and their local variables. During context switch, the stack is updated first by pushing the value of its own *thread pointer* (depicted in gray). The subsequent next position in the stack consistently opens a new stack frame (depicted in blue) for the thread process (*func1*). Since the stored thread context also contains the memory addresses, the current base pointer address (*rbp = addr A*) is recorded. After a few memory addresses, the previously identified base pointer (*rbp = addr A*) was pushed onto the stack, indicating that a new stack frame (denoted in orange) was created. Now, the memory address corresponding to this position becomes the new base pointer (*rbp = addr B*). Likewise, by identifying the chain of base pointers (*rbp*) it is possible to locate individual stack frames. The position one above the base pointer (*rbp*) pertains to the function return address (*ret*). The memory addresses in stack excluding this metadata (*rbp* and *ret*) gives the positions where the local variables of the functions are located. The last stack frame corresponds to QuickThreads, which stores the host register values and the pointer to the scheduler queue.

Checkpointing - Restore: The execution flow of the checkpoint library in restore mode is shown in Figure 3 (right). The same SystemC module is executed again with the checkpointing library in restore mode. Due to dynamic memory allocation, the memory regions of the newly instantiated module processes (stack addresses depicted in red, values in yellow) differ from those in the prior execution during store mode. After module instantiation, the SystemC kernel transfers control to the checkpointing library, which analyzes the checkpoint file to retrieve the thread simulation state (stack metadata and local variable positions) within each stack frame. The current thread simulation state (depicted in yellow) is updated with the simulation state from the checkpoint file (depicted in green). To maintain the integrity of the new stack region during restoration, only the stack memory region of the local variables are updated with the value from the checkpoint file and the thread specific data such as its stack pointer (*rsp*), stack base pointer (*rbp*), and function return address (*ret*) remain unaltered. Altering these could lead to memory corruption and failure of the SystemC kernel to manage threads. Once restoration is complete, control returns to the SystemC kernel, and the simulation resumes directly from the restored checkpoint state, ensuring continuity.

5.3 Checkpointing Illustration of the Running Example

This subsection describes the application of the proposed checkpointing methodology to the running example (shown in Listing 2).

5.3.1 Store

The SystemC counter module, compiled with the checkpointing library, is executed in store mode with a checkpoint simulation time of 10 µs. During the simulation, when the checkpoint condition is reached, the checkpoint library iterates through all instantiated objects of the wrapper class datatype and locates the *target_cnt_vh* tracking the module parameter *target_cnt*. Using the *serialize()* function defined within the wrapper class, the parameter's value is serialized and stored in the checkpoint file along with its hierarchical name (*top.counter.target_cnt = 20*).

After storing the module parameter, the state of the SC_THREAD process instantiated in the module is also stored. The checkpointing library iterates through all instantiated thread objects of type SC_THREAD process and identifies the *count_process()* thread object. Using QuickThreads function calls, details of the thread's stack memory (such as stack pointer, stack size, stack start address) are retrieved. The stack region of *count_process()* (illustrated in Figure 5) is then stored.

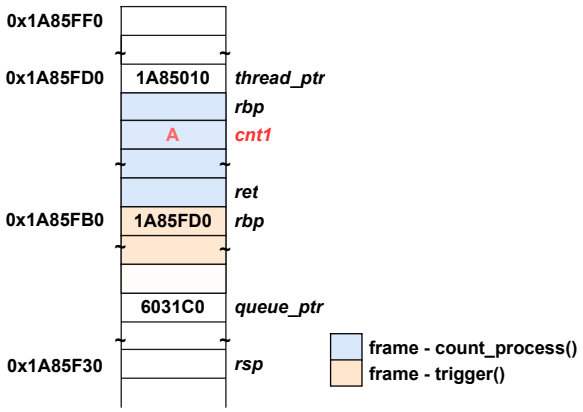


Figure 5 Counter Module Process Stack Layout - Store

The utilized stack region starts at the stack top address (0x1A85FF0) and extends to the stack pointer (0x1A85F30), with the stack growing from higher to lower memory addresses. This stack memory region, along with the memory addresses and the values held in them, is stored in the checkpoint file.

```
cnt1: 0
cnt1: 1
cnt1: 2
...
cnt1: 10
STORE THREAD:i_counter_mod1.count_process
```

Listing 3 Output of counter example with checkpointoing (save)

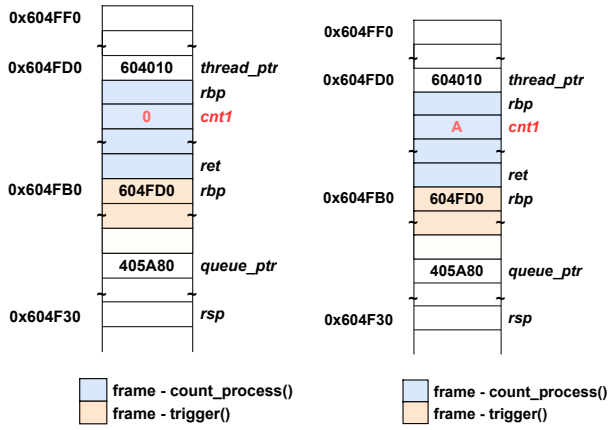
The console output of the counter module with checkpointing library executed in store mode is shown in Listing 3. The *cnt1* local variable starts counting until it reaches 10 and the checkpoint library is invoked to store the module state.

5.3.2 Restore

The SystemC counter module is executed with the checkpointing library in restore mode.
After the elaboration phase of the simulation (module instantiation), the checkpoint library is invoked to restore the counter module state. The checkpoint file is analyzed to retrieve the hierarchical module parameter name (*top.counter_mod.target_cnt = 20*), which is used to search for a matching parameter with the same hierarchical name in the current simulation. To locate the same module parameter in the checkpoint file, the library iterates through all instantiated module parameters. Since the same module is executed, an exact match for *target_cnt* is guaranteed, and its value is updated using the *deserialize()* function of the wrapper class. For instance, assigning a new value to *target_cnt* would adjust the counter to match the updated target. In this case, the *target_cnt* remains unchanged at 20.

The checkpoint file is further analyzed for restoring the simulation state of thread processes. Analyzing the stored stack memory region (illustrated in Figure 5) reveals two stack frame base pointers (0x1A85FD0 and 0x1A85FB0), corresponding to *count_process()* function (depicted in blue), and *trigger()* function (depicted in orange). After identifying the stack frame metadata, the memory region containing the local variable *cnt1 = 10* (depicted in red) is determined.

The simulation state of the thread processes before and after restore is illustrated in Figure 6. With the module process (*count_process*) being instantiated, its stack memory region is retrieved using QuickThreads (depicted in Figure 6a). In this simulation run, the stack memory region allotted for the thread process is different. The thread pointer is 0x604010 and its stack memory region starts from 0x604FF0 and its updated stack pointer is 0x604F30. The simulation execution has not yet begun, so the value of internal local variable *cnt1* is 0 (denoted in red). Then, the identified memory regions of the local variables in the stored checkpoint file are copied to the stack memory of the current process *count_process()* (depicted in Figure 6b). The *cnt1* local variable in



(a) Before Restore **(b)** After Restore

Figure 6 Counter Modue Process Stack Layout - Restore

the new thread stack memory that holds the value 0 is updated to the value of 10 (value of *cnt1* in checkpoint file).

After restoring the thread's stack memory, the SystemC kernel resumes simulation by referencing the scheduler queue to schedule the next thread for execution. The scheduler queue maintained by the kernel for the counter module simulation (restore mode) is depicted in Figure 7.

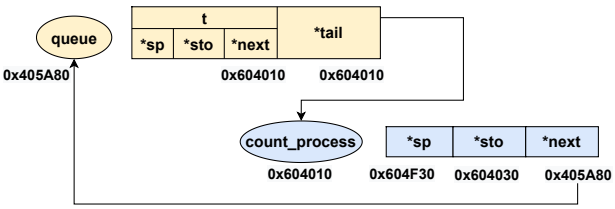


Figure 7 Data Structure of Scheduler Queue - Illustration for Counter Module
*sp: Pointer to Stack Top
*sto: Pointer to Base Stack Address
*next: Pointer to Next Execution Thread

QuickThreads, as previously discussed, facilitates context switching by transferring control to the next thread. It pushes the register values, switches thread and stores thread specific details, including the thread stack pointer of the old thread onto the new thread. When QuickThreads transfers control back to the old thread, this updated stack pointer is passed to it, so that it can retrieve the updated stack contents and resume its execution. This thread specific information is stored on a thread stack itself and the checkpointing library operates on the thread stack memory region which contain these addresses during the restore process. Hence, it is critical not to tamper with the addresses in the stack that hold these thread-specific values, such as the thread stack pointer (0x604F30). These values are subsequently passed to the scheduler queue to indicate the stack pointer of the next thread to execute. Similar to the thread stack pointer, other simulation-specific details such as the thread pointer (0x604010) and queue pointer are dynamically assigned and should be unchanged.

```
cnt1: 0
RESTORE THREAD: i_counter_mod1.count_process
cnt1: 10
cnt1: 11
...
cnt1: 20
execute trigger function
```

Listing 4 Output of counter example with checkpointoing (restore)

```
1  // System (DUV) Initialization
2
3  if (restore) {  //RESTORE
4    thread_restore(chkpt_file_name);
5    sc_core::sc_start();
6  }
7  else {    // STORE
8    sc_core::sc_start(chkpt_sim_time);
9    thread_store(chkpt_file_name);
10 }
11
12 SC_REPORT_INFO("sc_main", "simulation complete");
```

Listing 5 Checkpoint Library API

In this simulation run, the scheduler queue (0x405A80) (depicted in yellow) points to the next thread for execution in the queue, which is thread *count_process* (0x604010) (depicted in blue). The thread specific details such as its stack pointer (**sp*), base stack address (**sto*) and the next thread to be executed (**next*) are stored in a data structure to facilitate proper scheduling. Altering these addresses during restoration can result in the scheduler queue retrieving incorrect data, causing it to fail in invoking the intended thread for execution, thereby disrupting the simulation.

The console output of the simulation run in restore mode is shown in Listing 4. Initially, the modules are initialized and *cnt1* value is equal to 0. Then, checkpoint library restores the state of the module. Later, the simulation resumes with *cnt1* starting at 10 and incrementing to 20. In restore mode, the counter module executes for only half the simulation duration, before invoking the *trigger()* function.

5.4 Integration of Checkpoint Library with VP

The checkpoint library provides a minimal *Application Programming Interface* (API) for seamless integration with existing SystemC-based VP. The library abstracts the complexities of the checkpointing mechanism while enabling developers to store and restore the state of the simulation efficiently. The code snippet (Listing 5) demonstrates the additional lines of code needed for integrating the checkpointing library into an existing VP simulation. Developers only need to include the checkpointing library during compilation and invoke the provided API from the top module of the simulation.

thread_store(): This function (Line 9 in Listing 5) is responsible for saving the state of the simulation at a given checkpoint time. Developer provides the name of the checkpoint file where the current state (module parameter and module process) will be stored. In this example, after the simulation has executed, the checkpoint library is invoked to store the simulation state.
thread_restore(): This function (Line 4 in Listing 5) is responsible for restoring the state of the simulation from a given checkpoint file. It must be invoked before the start of the simulation run. The function reads the checkpoint file and restores the values of the module parameter and state of the module process to match the saved state.

6 Case Study

This section presents a case study showcasing the effectiveness of the proposed checkpointing methodology for a SystemC VP. The case study utilizes a digital SystemC model of a FIR filter, provided by our industrial partner (COSEDA [20]), to validate the proposed methodology.

6.1 Digital FIR Filter

Filters are the fundamental components of *Digital Signal Processing* (DSP) architectures, used to extract useful information from signals by removing undesired components such as noise. They play a crucial role in design of electronic components in the automotive sector, including radar systems, engine control and driver-assistance systems. An FIR filter operates on discrete input samples, applying a set of coefficients to compute a weighted sum of the input values. This process ensures linear phase response and stability, making FIR filters suitable for a wide range of applications in signal processing.

The SystemC FIR filter module is presented in Listing 6. The module includes member variable *taps*, an internal buffer for storing input signal data, and *coeff*, which holds the constant filter coefficients. The number of individual coefficients or weights of the filter, *FIR_TAPS* is equal to 32. The thread process *fir_filter_proc()* implements the FIR algorithm. Its primary task is to perform a *multiply-accumulate* operation on the input signal and the filter coefficients (illustrated from Line 30 to Line 32 in Listing 6). The *fir_filter* module operates as follows:
Input Signal Storage and Buffer Update: The input signal *data_in* is stored in the internal buffer and the buffer is shifted on every rising edge of clock signal.
Output Signal Calculation: The output signal *data_out* is calculated by *fir_filter_proc()* process as the weighted sum of internal buffer and the filter coefficients.
Continuous Processing: At each rising edge of the clock signal, the internal buffer is updated with the current input signal *data_in* and the *data_out* signal is recalculated. Thus, the output signal value is influenced by both the current and previous input signal values stored in the buffer, enabling the FIR filter to process the signal continuously and effectively.

```
SC_MODULE(fir_filter){
  sc_core::sc_in<double> data_i;
  sc_core::sc_out<double> data_o;
  sc_core::sc_in<bool> clk_i;
  sc_core::sc_in<bool> rst_i;
  void fir_filter_proc();
  SC_THREAD(fir_filter_proc);
    sensitive << clk_i.pos();
  private:
    double taps[FIR_TAPS]; // input signal buffer
    double coeffs[FIR_TAPS] = {};
    cos_sc_variable_handle<double[FIR_TAPS]>
        taps_vh = {taps, "taps_handle"}; //
        marked for checkpointing
};

void fir_filter::fir_filter_proc(){
  double data_i_tmp;
  while(1){
    wait(); // clock edge
    if (rst_i.read()) {
      for (int i = 0; i < FIR_TAPS; i++) {
        taps[i] = 0.0;
      }
    } else {
      double acc = 0.0;
      data_i_tmp = data_i.read();
      SHIFT: for (int i = (FIR_TAPS - 1); i >= 0;
          i--) {
        taps[i] = (i == 0) ? data_i_tmp : taps[i -
            1];
      }
      acc = 0;
      MAC: for (int i = 0; i < FIR_TAPS; i++) {
        acc += coeffs[i] * taps[i];
      }
      data_o.write(acc);
    }
  }
}
```

Listing 6 FIR Filter SystemC Module

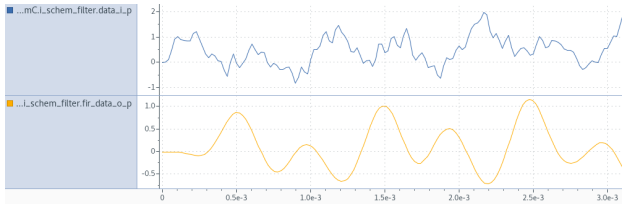


Figure 8 FIR Filter Output
Input signal (depicted in blue); Output signal (depicted in orange)

To illustrate the behavior of the FIR SystemC module, its output for a representative input signal is shown in Figure 8. The input signal (depicted in blue) is shown in the top section and the filtered output signal (depicted in orange) is shown in lower section. It can be inferred that the FIR module effectively removes the high frequency noise present in the input signal, with the output exhibiting a slight delay relative to the input.

With our proposed checkpointing library, the FIR module is restored from a saved state (result shown in Figure 9). Initially, the FIR module executes normally (denoted by orange graph) and is checkpointed at time 5 ms (indicated by the green dashed line). Subsequently, the module is executed with the same input signal in restore mode (denoted by blue graph), resuming execution from the saved checkpointed state. The sudden transition in the output signal (denoted by blue graph), indicates the instant when the restore operation occurred.

At the module elaboration phase of the simulation, the module parameter (*taps*) value is equal to 0. Its value only gets updated at the module execution phase after the checkpoint restore. Until then, the output signal holds the previous value and updates upon the first invocation of the module, resulting in a sudden spike at the beginning of the simulation. This updated module parameter ensures that the input buffer contains the same previous input signal values it would have held if the simulation had been executed continuously from the beginning. The simulation state of the thread process (*fir_filter_proc()*) is also restored during the checkpointing process. This ensures that the thread resumes execution exactly from the point where the checkpoint was created, with the next instruction of the process executed seamlessly after restoration. When the thread calculates the next output signal using the restored buffer, the resulting output value aligns with the expected output. This is because the output calculation inherently depends on the previous input values, which are accurately restored during the checkpointing process.

The results demonstrate that after restoring the state of the FIR module, it produces an output consistent with the expected behavior, as if the simulation had been executed uninterrupted from the beginning. This outcome highlights the reliability and effectiveness of the proposed checkpointing methodology in accurately restoring the simulation state of a SystemC digital VP.

7 Limitation and Future Work

A notable limitation is the handling of dynamically allocated memory inside SC_THREAD processes, due to its inherently complex non-deterministic address allocation, pointer dependencies, and potential mismatches during restoration. Similarly, checkpointing OS handles (e.g., file descriptors) pose challenges, as these handles often refer to external resources that cannot be easily recreated or restored in their original state, potentially leading to inconsistencies or resource conflicts. In highly optimized builds,the creation of frame pointers may be disabled, potentially impacting the checkpointing process. The stored checkpoint files are machine architecture-specific, requiring an initial simulation run on new machines to generate compatible files—a manageable step given the substantial advantages of checkpointing.

Future work will extend the proposed methodology to SystemC *Analog/Mixed-Signal* (AMS) models, addressing the challenges of mixed-signal designs to ensure reliable restoration. Currently, the methodology caters only to SC_THREAD process with a single *wait()* statement, and future efforts will focus on extending this capability to threads with multiple *wait()* statements.

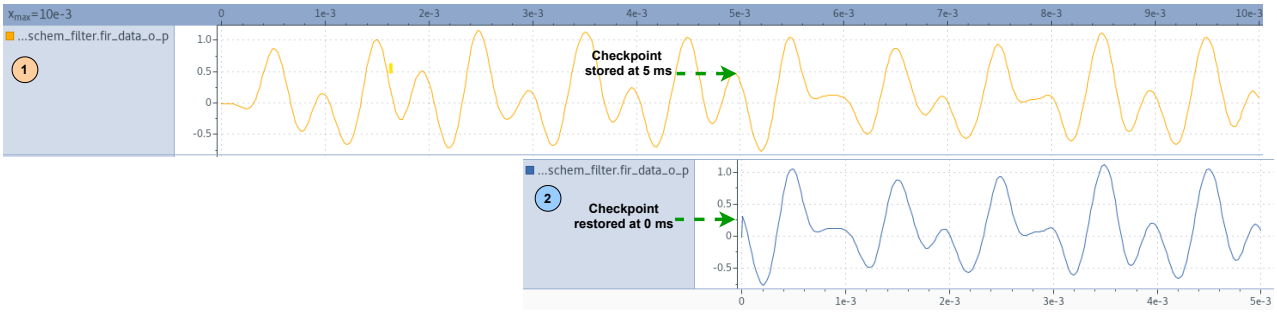


Figure 9 FIR Filter Restored using Checkpoint Library
1: Simulation run for 10 ms and FIR state stored at 5 ms
2: FIR state is restored and simulation run for 5 ms

Additionally, plans include enabling the checkpointing of SystemC signals to enhance the completeness and versatility of the methodology for complex simulation scenarios.

8 Conclusion

In this paper we presented a novel checkpointing methodology for SystemC digital VPs, with a particular focus on SC_THREAD processes. The methodology leverages the QuickThreads library to store and restore the simulation state of individual threads, without requiring any modifications to the SystemC kernel. The checkpointing methodology has been implemented as a SystemC library, ensuring seamless integration with existing VPs. The proposed methodology was demonstrated using a real-world SystemC module implementing FIR filter, showcasing its ease of applicability and simulation runtime benefits.

9 Acknowledgment

This work was supported by the German Federal Ministry of Education and Research (BMBF) within the project PaSVer under grant no. 16ME0855, the project ECXL under grant no. 01IW22002 and the project Scale4Edge under grant no. 16ME0127 and no. 16ME0135. Support from the LIT Secure and Correct Systems Lab funded by the State of Upper Austria is gratefully acknowledged.

10 Literature

[1] Industrial Task Force of edacentrum e.V., "Arbeitskreis automotive, working group virtual platforms, – white paper," 10 2019.

[2] T. De Schutter, *Better Software. Faster!: Best Practices in Virtual Prototyping*. Synopsys Press, March 2014.

[3] "IEEE 1666-2023 standard for standard SystemC language reference manual." [Online]. Available: https://doi.org/10.1109/IEEESTD.2023.10246125

[4] M. Hassan, D. Große, and R. Drechsler, *Enhanced Virtual Prototyping for Heterogeneous Systems*. Springer, 2022.

[5] V. Herdt, D. Große, and R. Drechsler, *Enhanced Virtual Prototyping: Featuring RISC-V Case Studies*. Springer, 2020.

[6] E. Roman *et al.*, "A survey of checkpoint/restart implementations," *Lawrence Berkeley National Laboratory, Tech. Citeseer*, vol. 5, 2002.

[7] "SystemC language working group (LWG) Accellera." [Online]. Available: https://www.accellera.org/downloads/standards/systemc

[8] D. Keppel, "Tools and techniques for building fast portable threads packages," 1993. [Online]. Available: https://api.semanticscholar.org/CorpusID:60785411

[9] J. Smith and R. Nair, "The architecture of virtual machines," *Computer*, vol. 38, no. 5, pp. 32–38, 2005.

[10] P. H. Hargrove and J. C. Duell, "Berkeley lab checkpoint/restart (blcr) for Linux clusters," in *Journal of Physics: Conference Series*, vol. 46, no. 1. IOP Publishing, 2006, p. 494.

[11] J. S. Plank, M. Beck, G. Kingsley, and K. Li, *Libckpt: Transparent checkpointing under UNIX*. Computer Science Department, 1994.

[12] M. Montón, J. Engblom, C. Schröder, J. Carrabina, and M. Burton, *Checkpoint and Restore for SystemC Models*. Dordrecht: Springer Netherlands, 2010, pp. 41–57.

[13] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A full system simulation platform," *Computer*, vol. 35, no. 2, pp. 50–58, 2002.

[14] S. Kraemer, R. Leupers, D. Petras, T. Philipp, and A. Hoffmann, "Checkpointing SystemC-based virtual platforms," *IJERTCS*, vol. 2, pp. 21–37, 10 2011.

[15] S. Kraemer, R. Leupers, D. Petras, and T. Philipp, “A checkpoint/restore framework for SystemC-based virtual platforms,” in *2009 International Symposium on System-on-Chip*.

[16] B. Farkas, “Standard compliant snapshotting for SystemC virtual platforms,” Ph.D. dissertation, Technische Universität Braunschweig, 2020.

[17] “Accellera SystemC initiative, universal verification methodology (uvm) 1.2 user’s guide,” Oct 2015. [Online]. Available: https://www.accellera.org/downloads/standards/uvm

[18] M. Jung, F. Schnicke, M. Damm, T. Kuhn, and N. Wehn, “Speculative temporal decoupling using fork(),” in *2019 Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2019, pp. 1721–1726.

[19] J. Vennin, S. Meftali, and J.-L. Dekeyser, “Understanding and extending SystemC user thread package to IA-64 platform,” in *Proceedings of International Workshop on IP Based SoC Design*, vol. 66, 2004.

[20] COSEDA Technologies GmbH, “Coside 3.2: The design environment for heterogeneous systems,” https://www.coseda-tech.com/coside-overview.