# Speculative Temporal Decoupling Using `fork()`

Matthias Jung[1], Frank Schnicke[1], Markus Damm[1], Thomas Kuhn[1] and Norbert Wehn[2]

[1] *Fraunhofer Institute for Experimental Software Engineering (IESE), Germany*
{matthias.jung, frank.schnicke, markus.damm, thomas.kuhn}@iese.fraunhofer.de
[2] *Technische Universität Kaiserslautern, Germany*
wehn@eit.uni-kl.de

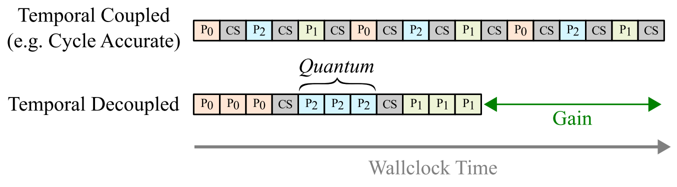
***Abstract*—Temporal decoupling is a state-of-the-art method to speed up virtual prototypes. In this technique, a process is allowed to run ahead of simulation time for a specific interval called quantum. By using this method, the number of synchronization points, i.e. context switches, in the simulator is reduced and therefore, the simulation speed can be increased significantly. However, using this approach can introduce functional simulation errors due to missed synchronization events. Thus, using temporal decoupling implies a trade-off between speed and accuracy and the size of the quantum must be chosen wisely with respect to the simulated application. In loosely timed simulations most of the functional errors are tolerable for the sake of simulation speed. However, for instance safety critical errors are rare but can lead to fatal results and must be handled carefully. Prior works present mechanisms based on checkpoints (storing/restoring the internal state of the simulation model) in order to rollback in simulation time and correct the occurred errors by forcing synchronization. However, checkpointing approaches are intrusive and require changes to both the source code of all the used simulation models and the kernel of the simulator. In this paper we present a non-intrusive rollback approach for error-free temporal decoupling, which allows the usage of closed source models by using Unix's `fork()` system call. Furthermore, we provide a case study based on the IEEE simulation standard SystemC.**

***Index Terms*—Temporal Decoupling, Fork, SystemC, Time Quantum, Virtual Prototyping**

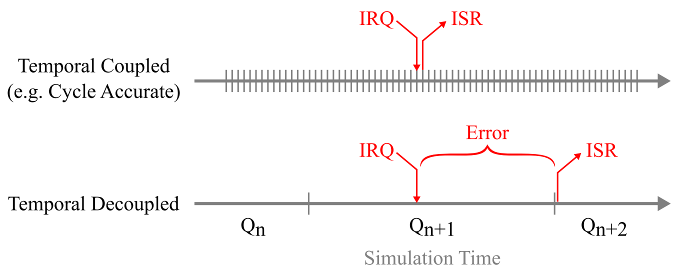
## I. Introduction

Nowadays, companies have to deal with complex hardware architectures and trends such as heterogeneous multi-core systems, that include CPUs, GPUs and custom accelerators. Therefore, there exists a constant pressure to adapt to these emerging trends and deliver their products quickly because of the competitive market [1]. Traditional design-flow procedures have a performance problem due to the high complexity of modern systems. Therefore, new tools and approaches for system design are needed to fulfill these requirements.

An effective approach for this issue are *Virtual Prototypes* (VPs), where gut feeling based engineering processes are replaced by evidences produced from system simulations in order to hedge decisions. In fact, virtual prototypes are fully functional software models of hardware, software and the physical environment, which are used for design space exploration because of easy modification and fast simulation speed. However, for VPs there exists a challenging trade-off between a fast simulation and an accurate simulation, which has to be considered wisely for the beneficial usage of VPs in the development flow.


Temporal Coupled
(e.g. Cycle Accurate)
P0 CS P2 CS P1 CS P0 CS P2 CS P1 CS P0 CS P2 CS P1 CS
Quantum
Temporal Decoupled
P0 P0 P0 CS P2 P2 P2 CS P1 P1 P1
Gain
Wallclock Time

(a) **Simulation Speedup Gain with Temporal Decoupling:** By grouping the execution of SPs in so called quanta, the simulation duration is reduced [4].



(b) **Accuracy of Temporal Decoupling:** Since in temporal decoupled simulations SPs can run ahead of simulation time, there exists the possibility of a functional error.

Fig. 1: Temporal Decoupling

State-of-the-art system simulators, such as SystemC TLM [2] or FERAL [3] support the concept of *Temporal Decoupling* as a dynamic control lever mechanism for this trade-off.

As shown in Figure 1a, in temporally coupled discrete event driven simulations (e.g. cycle accurate) *Simulation Processes*[1] (SPs) are synchronized at each event they are sensitive to by the simulation kernel (cf. delta-cycle [2]). The regular *Context Switch* (CS) between the different SPs ($P_0$, $P_1$ and $P_2$) possess a significant overhead in wall-clock time[2]. By using temporal decoupling, SPs are allowed to run ahead of simulation time[3] for a specific interval called *Quantum*, as shown in Figure 1a. The grouping of SP executions in quanta,

[1]For example: `SC_THREAD` and `SC_METHOD` in SystemC, `PROCESS` in VHDL, `ALWAYS` in Verilog, or `doStep()` in FERAL.
[2]The wall-clock time is the time that passes from the start of the execution to completion of the simulation for a human observer.
[3]The simulated time is the time being modeled by the simulation, which may be less than or greater than the simulation's wall-clock time.

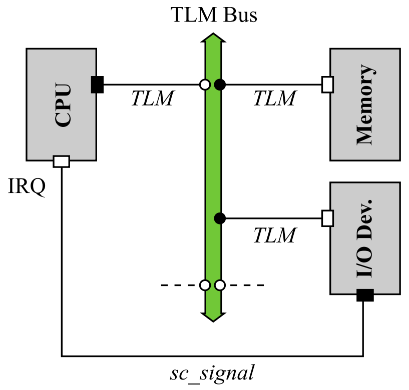


Fig. 2: **Example Platform**

which are executed atomically, decreases the total CS overhead and therefore the simulation duration is reduced.

However, by using this quantum mechanism functional simulation errors can occur. As an example, consider the system shown in Figure 2: If the I/O device generates an *Interrupt Request* (IRQ) for the CPU, the *Interrupt Service Routine* (ISR) will be called a few cycles later in a temporal coupled simulation. In contrast, in a temporal decoupled simulation, the CPU will register the IRQ event at the beginning of the next quantum for the first time because the current quantum was executed atomically. This might lead to a functional error, as shown in Figure 1b. Therefore, the concept of temporal decoupling is a comfortable mechanism to control the trade-off between accuracy and speed: If the quantum is small, the simulation is more accurate but slow. If the quantum is large, the simulation is faster but less accurate.

There exist scenarios where a fast simulation is required and most functional errors can be tolerated – nevertheless, some rare events like specific IRQs should be accurate in time, e.g. because they are safety critical. Another example of such a type of event is the reaction to (injected) transmission errors, which are very rare, but the correct simulation of the system's response to these errors is often very important. Further examples are coupled analog simulators or hardware-in-the-loop simulations, where asynchronous events could occur [4].

To address this, Gläser et al. [4] present the idea of using simulation *Checkpoints* [5], [6] in order to rollback in simulation time and force an earlier synchronization to correct the occurred errors. For the creation of a checkpoint, the state of all models in the system has to be extracted and stored. For example, the processor simulator gem5 [7] stores checkpoints on the hard drive, because draining the pipeline and serializing all the registers and memories in the system can require several gigabytes of storage [5]. Unfortunately, there exists no standardized concepts for checkpointing in SystemC, but, these are currently under discussion in the standardization committee [8].

However, checkpointing approaches are intrusive and require changes to both, the source code of all the used simulation models and the kernel of the simulator. Therefore, using checkpoints might be impossible because many legacy models have to be updated with checkpointing capabilities and many models exist only as closed-source precompiled libraries.

In this paper, we present a non-intrusive rollback approach by using Unix's `fork()` system call, which allows error free temporal decoupling and requires changing only the models of interest. The basic idea is to execute a quantum step speculatively and, in case of an error[4], rollback to a previously forked process (i.e. a clone of the simulation) where the error is corrected by forcing synchronization at the point in time where the error occurs. By using our approach, no changes in all simulation models or the simulation kernel are required, and closed source precompiled libraries can be used seamlessly. In order to demonstrate the feasibility, we compare two different implementations for SystemC and assess the results.

This paper is structured as follows: Section II presents the related work. Our contribution, the speculative temporal decoupling using `fork()`, is presented in Section III. We show an implementation case study in SystemC and present simulation results in Section IV. Section V discusses further applicability of the approach and concludes the paper.

## II. Related Work

Gläser et al. [4] present a predictive quantum control and propose checkpointing as a rollback mechanism. However, in this work they concentrate only on the prediction scheme and do not implement any rollback mechanism for the sake of simplicity. Traditional checkpointing approaches for virtual platforms are presented in [5], [6]. Engblom et al. summarized the advantages of checkpoints at the SystemC Evolution Day 2017 [8]:

(1) **Undo Target Action:** Get back to a previous good state after something went wrong.
(2) **Safe the Boot:** Avoid redoing work, save booted and configured system for reuse and distribution.
(3) **Parallelize Test Execution:** Save checkpoints and spin up additional parallel simulations during testing.
(4) **Gear Shifting:** Save state from a fast VP and transfer to a more detailed VP.
(5) **Report Software Bugs:** Save state of the system when bug hits, transfer to SW developer for analysis.
(6) **Safe for the Day:** Save current state, shut down simulation and continue the next day at the same point.
(7) **Report Model Bugs:** Provide HW and SW state when a model bug hits to model developer for analysis.

Brandner [9] presents a rollback mechanism for the precise simulation of IRQ based on LLVM. Weinstock et al. [10] presents parallel SystemC implementations with a flexible time decoupling. Approaches to avoid errors are presented in [11]–[13]. The authors of [14] show that causality errors can exists for coupling SystemC AMS with temporal decoupled SystemC TLM models. The authors of [15] propose a method for software debugging using `fork()` in order to roll back to a previous execution state for controlled debugging sessions on real hardware devices.

[4]For example, an error could be detected when an internal IRQ variable is set to true at the beginning of a new quantum.

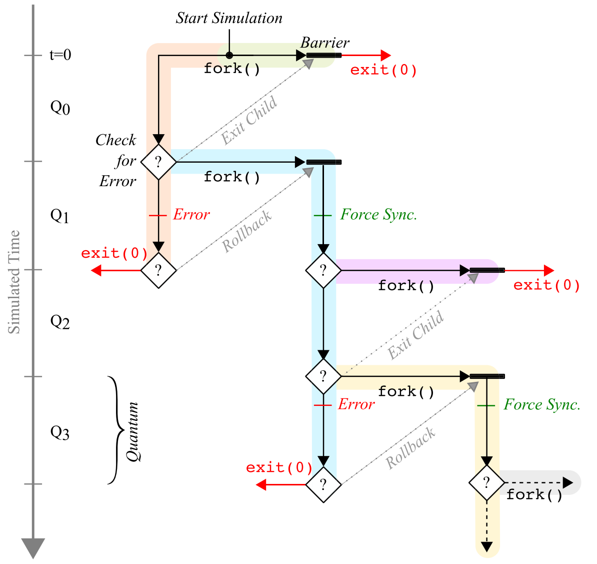
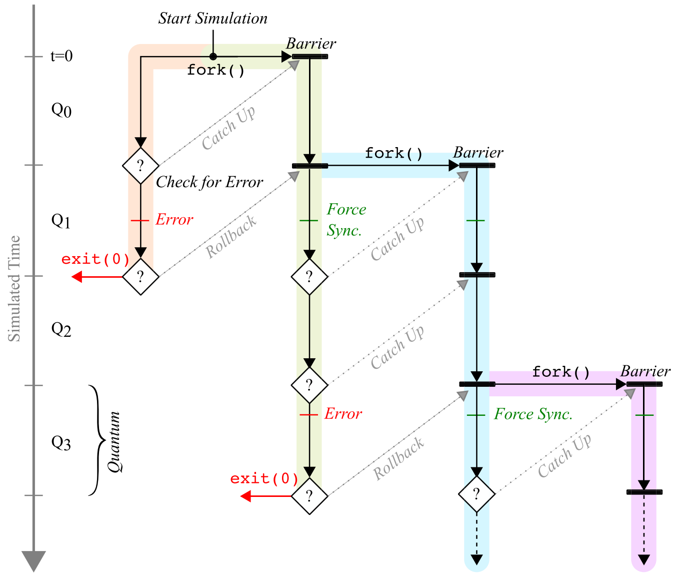
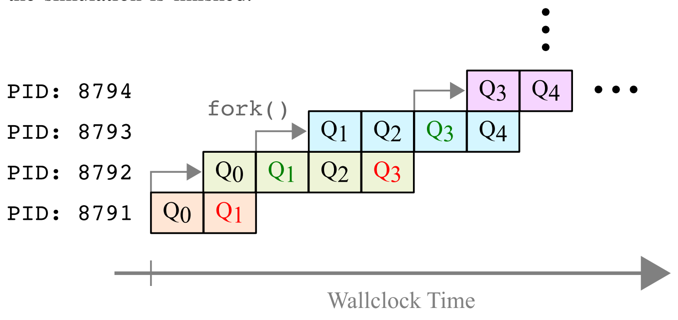


Fig. 3: **Naïve Implementation:** After each quantum, a `fork()` system call is done to create a backup of the simulation state. If no error happens, this backup is discarded. If an error happens, this backup is used to rectify the error and the main OSP is abandoned. This process is repeated until the simulation is finished.



(a) **Lock-Step Implementation:** The *main* OSP simulates a time quantum and checks for errors. If there is no error, the time shifted OSP can catch up. If an error happens, the *main* OSP reports this error and exits gracefully. The time shifted *backup* OSP is now the *main* process and forks itself and corrects the error. This is repeated until the simulation is finished.



(b) **Quantum Execution in OSPs Related to Wall-Clock time:** This visualizes the processes shown in Fig 4a. A red quantum denotes a simulation error while a green quantum denotes a recovery from this error. For the sake of simplicity, the forking overhead is not shown.

Fig. 4: **Lock-Step Fork**

## III. Speculative Temporal Decoupling

In the context of Unix-like operating systems such as *Linux*, *FreeBSD* or *macOS*, `fork()` is a system call that allows a process in the operating system to create a one-to-one copy of itself, called child. The fork operation creates a separate address space for the child, which has an exact copy of all the memory of the parent process. Today's modern operating systems are not duplicating the complete memory space of a process when `fork()` is called. Instead, a so called *Copy-on-Write* (CoW) semantic of the virtual memory model is applied. With this technique the copy operation is deferred to the first change i.e. write to a memory page. In other words, a memory page is only copied in the moment of a change, which makes `fork()` quiet efficient.

In our approach `fork()` is exploited as an elegant way to backup the state of a simulation efficiently, such that it is possible to execute the next quantum speculatively and in case of an error, to rollback in simulation time to an earlier state to re-simulate and correct an error.

To be able to distinguish *Simulation Processes* (SPs) from the processes created by `fork()` we call the latter from now on *Operating System Process* (OSP). For the sake of simplicity only one SP is considered in the following. The proposed approaches can be extended to multiple SPs by applying the described pattern to each SP. In the following, we present a *naïve* approach, which uses `fork()` for each quantum and a *Lock-Step* approach that uses `fork()` only if an error happens.

### A. Naïve Approach

Since errors can occur in any quantum, it must be possible to detect their occurrence and then to rollback the execution of a quantum in order to re-execute the simulation with a tighter temporal coupling. *The main idea of the naïve approach is to fork the simulation for each quantum execution to create a backup of the current simulation state. If no error happens, this backup is discarded. If an error happens, this backup is used to rectify the error.*

At the beginning of the simulation `fork()` is called. The current OSP is called the *main* OSP while the OSP child created by the `fork()` is called the *backup* OSP. Both OSPs are connected by the usage of a Unix `pipe`. Using this `pipe`, the *main* OSP sends commands to the *backup* OSP. Since reading from a `pipe` is blocking it acts like a barrier and the *backup* OSP waits until it receives a command.

There exist two different commands:

- **Exit**: No simulation error occurred during the simulation of one quantum in the *main* OSP. The *backup* process is not needed anymore and can exit.
- **Rollback**: A simulation error occurred in the *main* OSP. To rectify the error, the *backup* OSP has to re-simulate the last quantum with an earlier synchronization and the *main* OSP exits. The *backup* OSP is now considered as the new *main* OSP.

Figure 3 shows the simulation flow where errors are occurring during the quanta $Q_1$ and $Q_3$. In the case of a simulation error the *main* process sends the rollback command to the *backup* OSP with a hint at which point in time the error happened. Then the *backup* thread runs the failed quantum again and forces an earlier synchronization (e.g. calling `wait(hint)` in SystemC) or adjusting the length of the quantum dynamically as presented in [4]. This procedure is repeated until the simulation is finished successfully.

By using this naïve approach, it is possible to create an error free loosely coupled simulation. However, the drawback is the frequent usage of `fork()`. This usage introduces a significant overhead and thus slows down the simulation. Due to this, the speed benefit of using temporal decoupling vanishes as shown in Section IV. Consequently, we present in the next subsection a more sophisticated approach that reduces the number of `fork()` system calls significantly.

### B. Lock-Step Approach

As previously described, the drawback of the naïve rollback is the overhead of the frequent `fork()` system calls. This section proposes a more sophisticated approach, reducing the number of `fork()` system calls significantly and thus enabling a speed boost of simulations without any loss of accuracy.

*In contrast to the naïve approach, the idea of the Lock-Step approach is to call `fork()` only if an error occurs. Instead of forking after every quantum and abandoning the unneeded OSPs, the lifetime of a forked OSP is greatly increased.* To be able to do this, the forked OSP is used in a time shifted lock-step as shown in Figure 4. The *main* OSP simulates a quantum speculatively and checks for errors. If there is no error, the waiting *backup* OSP can catch up and simulates the current quantum again to reach the exact simulation state at the next quantum boundary. In the meantime, the *main* OSP is simulating the next quantum. Since the operating system will try to run both OSPs on two cores in parallel, the overhead of running the simulation twice can be neglected. If an error happens, the *main* OSP reports the error with a time hint to the *backup* OSP and exits gracefully. The time shifted *backup* OSP is now the *main* OSP and forks itself in order to have a new *backup* OSP. This procedure is repeated until the simulation is finished. Figure 4a shows the behavior of the different processes in time, visualizing the presented lock-step approach. Again, in this visualization errors are occurring in the quanta $Q_1$ and $Q_3$. To implement the lock-step approach, the `fork()` system call is called at the beginning of the simulation in order to create the time shifted *backup* OSP. Again, these OSPs communicate by the usage of a pipe. There are two different commands the *main* OSP can send to the time shifted *backup* OSP:

- **Catch Up**: No simulation error occurred during the time quantum. The time shifted *backup* OSP can safely catch up while the *main* OSP performs the next quantum.
- **Rollback**: A simulation error occurred in the *main* OSP. To rectify it, the *backup* OSP has to re-simulate the last quantum and the *main* OSP exits. The *backup* OSP is now considered the *main* OSP and forks itself.

Since the pipe acts as a FIFO queue, there is no limit on the number of error free time quanta the leading *main* OSP can hurry ahead while the time shifted *backup* OSP is catching up.

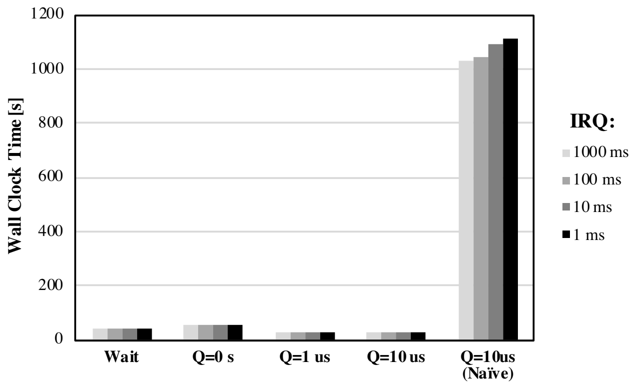
This method reduces the number of `fork()` system calls in comparison the the naïve approach by only using `fork()` in the case of an error. This can significantly reduce the overhead and thus allows a speed boost of the simulation, as shown in the next section.

## IV. A Case Study with SystemC TLM

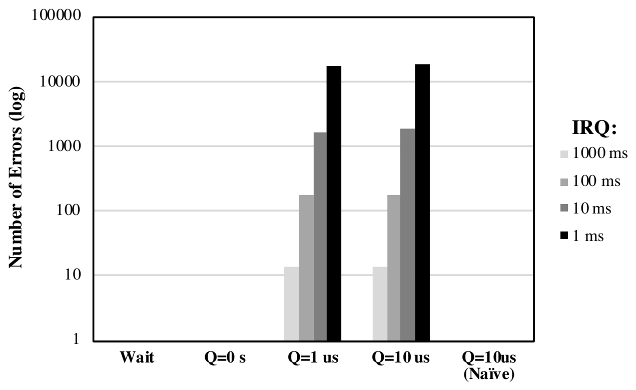
In order to demonstrate the feasibility of our approach, we compare the naïve and the lock-step approach in SystemC TLM. As an example platform we use the system shown in Figure 2. We use abstract TLM models using the loosely-timed coding style. The CPU initiator creates memory accesses to the main memory regularly by using `b_transport()` semantics. The I/O device generates interrupts in different rates (ranging from 1 ms to 1 s) for the CPU initiator, which runs the ISR that communicates with the I/O device. The system is simulated for 20 s simulation time. The simulations are executed on a MacBook Pro with a 2.8 GHz Intel Core i7 and 16 GB and 1.6 GHz DDR3 Memory running macOS High Sierra (10.13.6). Each simulation configuration has been executed three times and is averaged. In the first part of the case study we present the high overhead of `fork()` system calls in the naïve approach. In the second part we show the promising lock-step approach where the number of forks is reduced significantly.

### A. Naïve Approach

Figure 5a shows the wall-clock time i.e. the simulation duration: As baseline no temporal decoupling is used, where the simulation is synchronized with `wait()` statements. In order to assess temporal decoupling, quantum sizes of 0 s, 1 $\mu$s, 10 $\mu$s are used. The figure shows the run-time of the naïve approach for a quantum size of 10 $\mu$s. Even though the errors are fully eliminated, as shown in Figure 5b, the simulation is two orders of magnitude slower than the base line, which uses no temporal decoupling. This clearly shows that the naïve approach is not applicable due to the large overhead of `fork()` for each quantum. Since this approach is not applicable we omit implementation details and concentrate on the following lock-step approach.

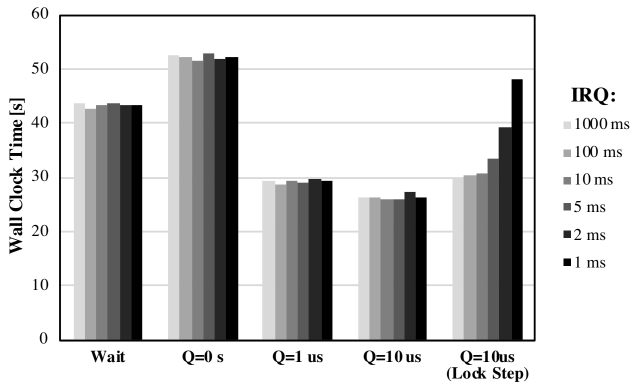


(a) **Simulation Duration:** The wall-clock time is assessed in the relation to the synchronization method and the frequency of the IRQ

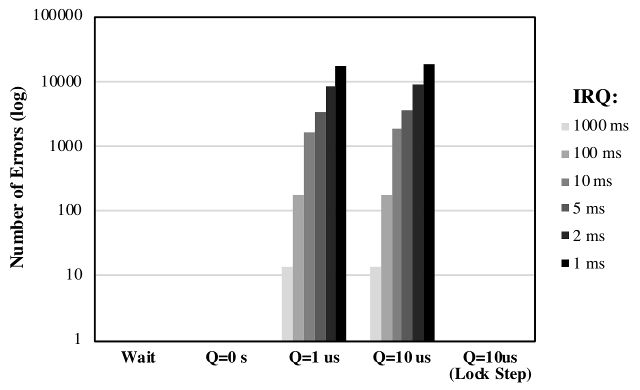


(b) **Number of Errors:** The error frequency is assessed in relation to the synchronization method and the frequency of the IRQ

Fig. 5: Results for Naïve Approach



(a) **Simulation Duration:** The wall-clock time is assessed in relation to the synchronization method and the frequency of the IRQ.



(b) **Number of Errors:** The error frequency is assessed in relation to the synchronization method and the frequency of the IRQ.

Fig. 6: Results for Lock-Step Approach

## B. *Lock-Step Approach*

Listing 1 shows a simplified implementation of the lock-step approach. The function `doFork()` has to be called before the calling of `sc_start()` in the `sc_main()` function to fork the simulation at simulation time zero. In the initiator CPU, that receives the critical IRQ signal, the function `childWait()` has to be called in order to block the *backup* OSP until the *main* OSP notifies the child over the `pipe`. If the *main* OSP detects an error at the beginning of the quantum (e.g. because the internal IRQ variable is set to true) it notifies and unblocks the *backup* OSP by calling `rollback(hint)`, where `hint` denotes the time where the IRQ happened. The *backup* process gets unblocked and can force an earlier synchronization e.g. by setting a new quantum value according to the `hint`. If no error occurs the *main* OSP calls the function `catchUp()`, the *backup* OSP gets unblocked and catches up with the simulation time.

Figure 6a shows the simulation duration for the lock-step approach. Again, the full synchronization with `wait()` is considered as baseline. Using a quantum of $0\,\text{s}$ shows the overhead of using the quantum mechanism in SystemC with a tight synchronization. If a usual quantum size of $10\,\mu\text{s}$ is used, a clear speedup of up to $\times 1.65$ can be observed. However, as shown in Figure 6b the simulation is not error free. Using the lock-step approach at a quantum size of $10\,\mu\text{s}$ for moderate IRQ frequencies ($1\,\text{s}$, $100\,\text{ms}$, and $10\,\text{ms}$) shows a speedup of $\times 1.43$ and is therefore 15% slower than the plain temporal decoupling with a quantum size of $10\,\mu\text{s}$. However, in this approach no simulation errors are occurring, as shown in Figure 6b, because all of them could be corrected. If the accuracy for the high interrupt rates is required, then it is more beneficial to not use temporal decoupling and using a regular coupled simulation using `wait()` statements. These results show the feasibility of our approach for rare but important events.

## V. Conclusion and Future Work

In this paper we presented a non-intrusive rollback approach for error free temporal decoupling, by using Unix's `fork()` system call. Furthermore, we provided a case study based on SystemC and show that with a slight overhead an error free simulation can be achieved. Especially for the simulation of safety critical IRQs, transmission errors etc. this approach is beneficial because it can be mixed with non-critical events.

Therefore, it is an elegant way to master the accuracy-speed trade-off in temporal decoupled simulations. With respect to Engblom's presented advantages of checkpoints shown in Section II, our presented approach using `fork()` can be used additionally for the advantages (3), (4) and (5). In the future we plan to extend the implementation for multiple errors in one quantum.

## ACKNOWLEDGMENT

This work was supported by the the *Fraunhofer High Performance Center* for *Simulation- and Software-based Innovation*. The authors thank Synopsys for their support.

## REFERENCES

[1] T. De Schutter, *Better Software. Faster!: Best Practices in Virtual Prototyping*. USA: Synopsys Press, 2014.
[2] IEEE Computer Society, "IEEE Standard for Standard SystemC Language Reference Manual," no. IEEE Std 1666-2011, 2012.
[3] T. Kuhn, T. Forster, T. Braun, and R. Gotzhein, "FERAL - Framework for simulator coupling on requirements and architecture level," in *2013 Eleventh ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2013)*, Oct 2013, pp. 11–22.
[4] G. Glaser, G. Nitschey, and E. Hennig, "Temporal decoupling with error-bounded predictive quantum control," in *2015 Forum on Specification and Design Languages (FDL)*, Sept 2015, pp. 1–6.
[5] 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*, Oct 2009, pp. 161–167.
[6] M. Monton, J. Engblom, and M. Burton, "Checkpointing for Virtual Platforms and SystemC-TLM," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 21, no. 1, pp. 133–141, Jan 2013.
[7] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011. [Online]. Available: http://doi.acm.org/10.1145/2024716.2024718
[8] J. Engblom, "Virtual Platform Checkpointing (SystemC Evolution Day 2017)," https://software.intel.com/en-us/blogs/2017/09/12/virtual-platform-checkpointing-systemc-evolution-day-2017, 2017.
[9] F. Brandner, "Precise Simulation of Interrupts Using a Rollback Mechanism," in *Proceedings of th 12th International Workshop on Software and Compilers for Embedded Systems*, ser. SCOPES '09. New York, NY, USA: ACM, 2009, pp. 71–80. [Online]. Available: http://dl.acm.org/citation.cfm?id=1543820.1543833
[10] J. H. Weinstock, R. Leupers, and G. Ascheid, "Parallel SystemC simulation for ESL design using flexible time decoupling," in *2015 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, July 2015, pp. 378–383.
[11] O. Bringmann, W. Ecker, A. Gerstlauer, A. Goyal, D. Mueller-Gritschneder, P. Sasidharan, and S. Singh, "The next generation of virtual prototyping: Ultra-fast yet accurate simulation of HW/SW systems," in *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2015, pp. 1698–1707.
[12] S. Stattelmann, O. Bringmann, and W. Rosenstiel, "Fast and accurate resource conflict simulation for performance analysis of multi-core systems," in *2011 Design, Automation Test in Europe*, March 2011, pp. 1–6.
[13] S. Hufnagel, "Towards the Efficient Creation of Accurate and High-Performance Virtual Prototypes," doctoralthesis, Technische Universität Kaiserslautern, 2014. [Online]. Available: http://nbn-resolving.de/urn:nbn:de:hbz:386-kluedo-38928
[14] M. Damm, C. Grimm, J. Haas, A. Herrholz, and W. Nebel, "Connecting SystemC-AMS models with OSCI TLM 2.0 models using temporal decoupling," in *2008 Forum on Specification, Verification and Design Languages*, Sept 2008, pp. 25–30.
[15] S. M. Srinivasan, S. Kandula, C. R. Andrews, Y. Zhou *et al.*, "Flashback: A lightweight extension for rollback and deterministic replay for software debugging," in *USENIX Annual Technical Conference, General Track*. Boston, MA, USA, 2004, pp. 29–44.

Listing 1: Simplified Implementation of the Lock-Step

```
class lsControl {
private:
    int pipefd[2];
    pid_t childPid;
    ...
public:
    ...
    void doFork() {
        if (pipe(pipefd) == -1) { // Create the pipe
            cout << "ERROR: Pipe" << endl;
            exit(-1);
        }
        childPid = fork(); // Do a fork

        if(childPid == 0) { // Backup OSP
            close(pipefd[1]);
            // Continue as usual ...
        }
        else if(childPid > 0) { // Main OSP
            close(pipefd[0]);
            // Continue as usual ...
        } else {
            cout << "ERROR: Fork" << endl;
            exit(-1);
        }
    }

    void catchUp() {
        if(childPid > 0) { // Main OSP
            char buf = 'c';
            write(pipefd[1], &buf, 1);
            // Continue both ...
        }
    }

    void rollback(sc_time hint) { // Simulation error
        if(childPid > 0) { // Main OSP
            char buf = 'f';
            write(pipefd[1], &buf, 1);
            write(pipefd[1], &hint, sizeof(sc_time));
            close(pipefd[1]);
            exit(0);
        }
    }

    sc_time childWait() { // Returns hint
        if(childPid == 0) { // Backup OSP
            char buf;
            // Blocking, child waits here
            read(pipefd[0], &buf, 1);
            if(buf == 'c') { // Catch up
                // Continue as usual ...
                return SC_ZERO_TIME;
            } else if(buf == 'f') {
                // Backup OSP forks itself and
                // becomes main OSP
                sc_time hint;
                read(pipefd[0], &hint, sizeof(sc_time));
                close(pipefd[0]);
                doFork();
                if(childPid == 0) { // New Child waits
                    return childWait();
                } else {
                    return hint;
                }
            } else {
                cout << "ERROR: Pipe Read" << endl;
                exit(-1);
            }
        }
        return SC_ZERO_TIME;
    }
};
```