Software-defined Temporal Decoupling in Virtual Platforms

1st Lukas Jünger
RWTH Aachen University
Aachen, Germany
juenger@ice.rwth-aachen.de

2nd Alexander Belke
RWTH Aachen University
Aachen, Germany
belke@ice.rwth-aachen.de

3rd Rainer Leupers
RWTH Aachen University
Aachen, Germany
leupers@ice.rwth-aachen.de

***Abstract*—As developers struggle with the ever rising complexity of HW/SW systems, the demand for high-performance full system simulators increases. So called virtual platforms, built using SystemC TLM 2.0, have tried to fill that need for early software verification and HW/SW co-design. Unfortunately, the increasing system complexity has not left the simulator's performance untouched. As they are sequential, adding more simulation components slows simulation execution. Temporal decoupling was introduced to satisfy the requirement of ever higher simulation speed by sacrificing some simulation accuracy. This is implemented by allowing simulation components to run ahead of the global simulation time by a static, predefined amount of time called *quantum*. The standardized SystemC TLM 2.0 static quantum approach however, does not lead to general performance improvements in all scenarios. One of the main reasons for this is, that the requirements towards the temporal decoupling strategy change over the simulation's duration and depend strongly on the executed target software. Therefore, a novel adaptive temporal decoupling technique is proposed in this work, that takes these requirements into account. This is achieved by non-invasive runtime profiling of the simulation and a later optimization of the temporal decoupling strategy using the gathered information. As shown in the case study of this work, the technique allows for performance increases of up to 5.87x compared to SystemC's static quantum approach in the presented benchmarks.**

***Index Terms*—Electronic System Level, Transaction Level Modeling, SystemC, Temporal Decoupling**

I. Introduction

Embedded systems have become ubiquitous, more powerful, and smaller over the past decades and they are evolving at a fast pace. Therefore it has become increasingly important to verify their correct functionality early in the design cycle. SystemC TLM 2.0 full system simulators, so called Virtual Platforms (VPs), are the standard tool for target software verification [1]. Making a VP available early, allows software development to start months before the actual hardware is available. As the VP and the hardware design can evolve in parallel, HW/SW co-design is enabled.

However, as system complexity rises so does the complexity of the VP. Because the SystemC TLM 2.0 simulation kernel is sequential, this rise in complexity leads to reduced simulation performance. Temporal decoupling is a common simulation acceleration technique to address this issue. Here, parts of the simulation are allowed to run ahead of the global simulation time by a predefined amount of simulation time called *quantum*. This quantum is usually manually selected and kept static over the entire simulation duration. The quantum may also be given in clock cycles. When the temporally decoupled component has reached the end of its quantum, it synchronizes time with the rest of the simulation and afterwards starts a new quantum. As the temporally decoupled simulation component has to be invoked less frequently to cover the same amount of simulation time, context switching overhead is reduced and performance can be increased. However, as the component runs ahead of simulation time, simulation events might occur at a point in simulation time, which the component has already surpassed. They are therefore seen too late or missed entirely, which introduces inaccuracy. By selecting a large quantum duration, performance can be maximized, but a larger quantum duration also results in a larger temporal error. Thus, a performance-accuracy trade-off has to be made, when using temporal decoupling. An example for this is depicted in Fig. 1. Here temporal decoupling is deployed for a CPU core. A static quantum size of 200 cycles is selected. It can be observed that there are incoming simulation events to the CPU core during quantum execution, for example interrupts. As the CPU is temporally decoupled, it sees those events at the beginning of its next quantum, when time is synchronized with the rest of

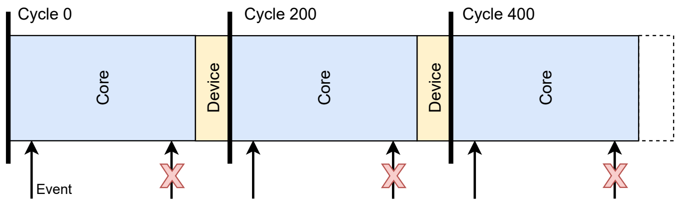


Fig. 1: Inadequately handled events with static quantum size

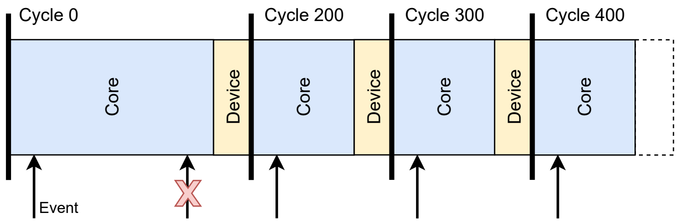


Fig. 2: Adaptive quantum size to handle events in time

the simulation. This leads to a loss in simulation accuracy that can manifest itself by interrupt handlers being triggered later than in a cycle-accurate simulation. If this is acceptable depends on the application being simulated. Finding the perfect quantum duration is a laborious and difficult task. Different parts of the application might profit from different quantum sizes and it is not always possible to find an option that fits all of them. Therefore, setting a static quantum duration for the entire simulation often leads to suboptimal compromises.

To overcome these issues we propose a target software behavior-driven temporal decoupling approach in this work. In our approach, the quantum size is adjusted during simulation execution to improve simulation performance by adapting to the requirements of the target software. This is depicted in Fig. 2. Here the quantum size is reduced so that fewer events are handled too late in the temporally decoupled component. The contributions of this work are:

- A novel adaptive temporal decoupling approach based on the behavior of the target software
- A representative case study comparing our approach with the SystemC static quantum

II. Related Work

Increasing simulation performance in VPs while retaining as much simulation accuracy as possible, has been an actively researched topic for many years. In this section, related work is summarized and compared to our approach.

Damm et al. implement a TLM-2.0 model with temporal decoupling that is connected to a SystemC-AMS model [2]. In their work, the AMS model acts as a streaming data producer or consumer. The TLM simulation is responsible for processing or generating this data. Their observation is, that for long quantum durations, there is an increasing amount of errors like data packet twists, however the performance increases due to a decreasing amount of context switches. They concluded, that for their application temporal decoupling implies a trade-off between accuracy and speed.

Gläser et al. consider a similar case and describe a method to adapt the quantum to a suitable length, so all packets can be processed [3]. They view the analog signals as asynchronous events and predict their occurrence using an adaptive filter. A safety margin is subtracted from the predicted time and the simulator is switched to a cycle-accurate mode until the event is triggered. Here the performance-accuracy trade-off is made by adjusting the size of the safety margin. This solution is optimized for one application whose performance is fully determined by the rate of received packets. Our approach is not tailored to a single application scenario and is therefore more generally applicable.

In [4] the authors analyze a dynamic quantum approach that sets the quantum duration to the time of the next timed event notification. This way no timed event notification can be missed or handled too late. When a simulation component generates many timed events, the average quantum size of other components can be decreased resulting in diminished performance. This occurs, because the SystemC timed event queue has global scope over all simulation components. An optimization is proposed, that takes into account whether a timed event notification is relevant to the component whose quantum is limited by it. If the event is not relevant, it can safely be ignored leading to longer average quantum durations and thus increased performance. The introduced approach considers only event notifications during simulation execution. Target software behavior is not examined, even though the authors acknowledge its performance influence.

A solution that avoids the issue of missed events is proposed by Jung et al. [5]. Here `fork()` is used to implement a quantum rollback mechanism. If an event is missed during quantum execution, the mechanism is used to re-execute this quantum. The quantum size is reduced accordingly to avoid missing the event. Correct execution is ensured, but an overhead is incurred by executing `fork()` for each quantum. The approach is focused on correct execution instead of performance. In our method, a more balanced approach is taken to find a good trade off between the two.

Another way of increasing performance is the parallelization of SystemC simulations. Weinstock et al. propose several methods for achieving near linear speedups. In [6] simulation objects are assigned to specific threads that are executed in parallel. A custom simulation kernel is used. Temporal decoupling is supported between threads, however determining the optimal quantum duration is again left to the developer. A similar approach is taken in [7]. Here, a parallelization framework for standard SystemC kernels is proposed, but similar limitations apply.

Yet another approach is presented by Stattelmann et al. in [8]. In their work the target software is compiled for the simulation host, so no expensive instruction set simulation is required. Binary annotation is used to capture the timing characteristics of the target. However, some simulation accuracy is given up compared to using an Instruction Set Simulator (ISS) as in our approach. In [9] the authors describe how a full multi-core system can be simulated with host-compiled target software.

III. Behavior-driven Temporal Decoupling

Choosing the quantum size aptly is important for performance and accuracy of the entire VP. However, this is a not a trivial task, as many factors influence the VP's temporal behavior, such as the implementations of the VP's models, the combination of models and the interaction between them, as well as the target software's behavior and its interaction with the models. Different phases of target software execution require different quantum sizes to perform optimally. In general, compute intensive phases require a large quantum, so that the computation can be finished quickly without many interruptions that incur context switching overhead. On the other hand, phases with lots of interaction with the virtual hardware, require a small quantum, so that the CPU of the VP can service interrupts quickly. A small quantum also allows other VP components to execute more frequently which improves interaction with physical systems outside the VP.

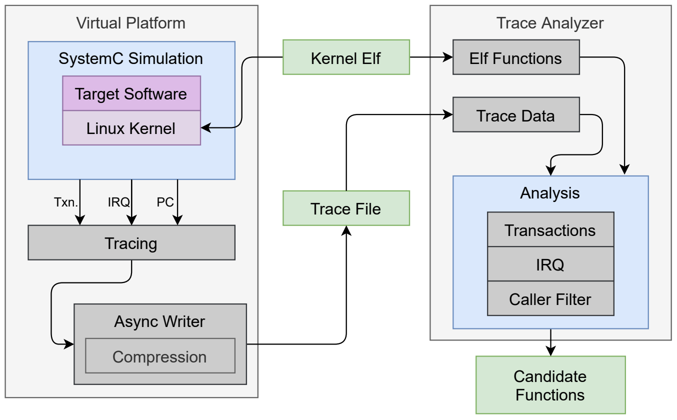


Fig. 3: Generation and analysis of tracing data

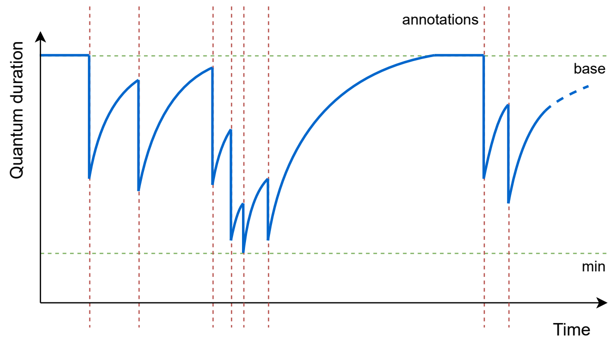


Fig. 4: Adaptation of quantum size with target software annotations

When using a static quantum size, as proposed in the SystemC TLM 2.0 standard, only a suboptimal compromise is achieved.

Therefore we propose an adaptive quantum approach, in which the quantum size is adjusted during simulation to the requirements of the target software. To achieve this, a two-step methodology is proposed. In a first step, the behavior of the target software is analyzed while it is executed on the VP. Trace data is collected during a profiling run and examined using a trace analysis tool. In a second step, the insights gained from the analysis are employed to optimize the quantum size during simulation execution to improve performance.

A. Behavior Analysis

An overview of the behavior analysis flow is shown in Fig. 3. To analyze target software behavior, its interactions with the VP are traced and recorded to a trace file during a profiling run. Our tracing approach is non-invasive and needs no target software modification. An ELF file of the target software with debug symbols is required. Ideally, source code should be available for inspection but it is not necessary. As many embedded systems execute Operating Systems (OSs), such as the Linux kernel, our approach focuses on this scenario, but is not limited to it. Here, the target software can be divided into two parts: the applications running in user space and the kernel running in kernel space. Applications access hardware by issuing system calls to the kernel, which invokes the device driver in accordance with the requested action. Interaction between the driver and hardware is facilitated using Memory-Mapped Input/Output (MMIO) and interrupts.

In VPs, MMIO is modeled using the blocking transport interface, while interrupts are implemented as `sc_signals`. Both are traced by a tracing component. To correlate them with target software execution, the Program Counter (PC) is traced as well. To reduce tracing overhead, the data is processed by a separate trace file writer thread. Since the tracer produces large amounts of data, on-the-fly compression is employed. Tracing the PC for every instruction is expensive, thus execution is only traced at basic block level, which is sufficient for reconstructing target software execution.

Once trace file generation is finished, it is processed by the trace analysis tool. In addition to the trace file, the ELF file of the target software is needed to lookup the functions corresponding to the recorded PCs. In the analysis step, several metrics are calculated. For each function in the ELF file, the total number of induced TLM blocking transport transactions is computed. This is used to weigh functions according to the amount of MMIO interaction they generate. Interrupts are traced to identify functions that are relevant for interrupt handling. For this, target software execution is analyzed over all executed quanta. If an interrupt is pending in the CPU model at the beginning of a quantum, the interrupt handlers are executed. To identify interrupt relevant functions, the number of times a function is executed during quanta with and without a pending interrupt is compared. If a function is executed more often in quanta with a pending interrupt, it is deemed relevant.

When handling the interrupt, the generic OS interrupt handlers are invoked, which then call device-specific interrupt handlers. As these generic handlers are also deemed relevant by the metric above, the analysis tool offers an automatic caller function filter. This filter inspects the call stack of the invoked functions to reduce the amount of misclassified functions. A function is deemed more relevant if it has no or a low number of interrupt relevant functions in its call stack.

Once analysis is complete, functions with a high interrupt or transaction activity are written into a candidate function file, which is used in the second step of our method to optimize the VP's performance.

B. Optimized Temporal Decoupling

After a list of candidate functions is generated, it is used to optimize the VP's performance by adjusting the quantum size to the requirements of the target software adaptively during simulation. A qualitative graph of this approach is shown in Fig. 4. Simulation is started at a base quantum q_{base}. The quantum duration is reduced when predefined annotation points in the target software are reached. These are the candidate functions from the previous step. With each reached annotation point, the quantum duration is reduced by

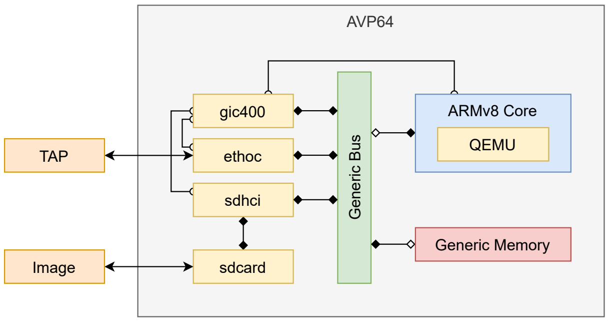


Fig. 5: Overview of AVP64 used for case study

a factor of $A \in [0,1]$. A lower bound to the quantum duration is defined as q_{min}:

$$q_{next} = \begin{cases} A * q_{cur.}, & \text{if} > q_{min} \\ q_{min} \end{cases}$$

The quantum size is increased slowly with every quantum that is executed without an annotation point being reached. The speed at which the quantum size is increased towards q_{base} is determined by a factor $B \in [0,1]$, by which the difference between the current quantum size and q_{base} is multiplied. In addition, a constant C can be used to define a minimum step size:

$$q_{next} = \begin{cases} B * (q_{base} - q_{cur.}) + q_{cur.}, & \text{if}(q_{next} - q_{cur.}) > C \\ C + q_{cur.} \end{cases}$$

In phases of high hardware interaction annotation points will be reached often, leading to small quantum durations, which increases the VP's accuracy. In compute intensive phases, few annotation points are reached. This leads to a rise in quantum size, increasing compute performance. In summary, increase and decrease are working against each other to yield an optimized quantum duration depending on the execution phase of the target software.

For specifying the annotation points, two options are supported. The first option is the insertion of annotation points into the target binary using semihosting instructions. This is useful for target software that is relocated during execution. The instructions can be added directly to the binary without source code or recompilation. However, this is problematic if the executable is cryptographically verified during execution.

To address this, the second option provides an automatic, non-invasive annotation method. Here, annotation points are handled by the ISS of the VP's CPU model. For this, the address of the candidate function is extracted from the ELF file. When the ISS reaches an annotation point, a callback is executed triggering the adaptive quantum mechanism.

IV. Case Study

A state-of-the-art VP was extended to include the required functionality and two benchmarks were undertaken to study the performance of our approach. All experiments were conducted on a host powered by an Intel Core i5-8250U with 16 GB of RAM running Linux 5.9. The *AVP64*[1] VP was used as a basis for the experiments. An overview of the VP is provided in Fig. 5. It consists of an ARMv8 processor model based on QEMU [10]. In addition it includes several peripherals such as an OpenCores ethoc Ethernet controller model, which is connected to the host through a Linux TAP device. An SDCard, connected to an SDHCI, holds the root file system of the Linux target software. The peripherals are connected to a CoreLink GIC-400 interrupt controller model using SystemC signals. The bus connections are realized with TLM sockets. As target software Linux 4.19 was used. GCC 10.2.0 was used for compilation on the host, while target software was compiled using GCC 7.5.0 for Aarch64.

The first benchmark uses the iperf3 utility[2] to measure TCP throughput to and from the VP. In this scenario iperf3 is started on the host and in the VP. To ensure correct calculation of the bandwidth against wall-clock time, measurements are taken on the host side. The VP's ethoc Ethernet controller uses Direct Memory Access (DMA) to process Ethernet frames. Ethoc's DMA buffer can store multiple frames, so the CPU can run other tasks, while ethoc is processing frames. The DMA is modeled by two SystemC threads, one for receiving and one for sending data. Each thread is executed once per quantum, so one packet can be sent or received. The completion of packet transfers is signaled via interrupt.

Function	IRQ factor	Transactions
ethoc_poll	2.07555	70548
ethoc_interrupt	13.8666	17143
gic_handle_irq	13.7583	9770
el0_svc_handler	2.04372	6933
__vfs_read	2.54092	6410
inet_recvmsg	2.38733	6026
add_interrupt_rand.	13.8189	5511
lock_sock_nested	3.02146	5160
__handle_domain_irq	12.6598	4724
generic_handle_irq	13.7921	4712
handle_fasteoi_irq	13.7965	4679
sock_read_iter	2.38733	4673

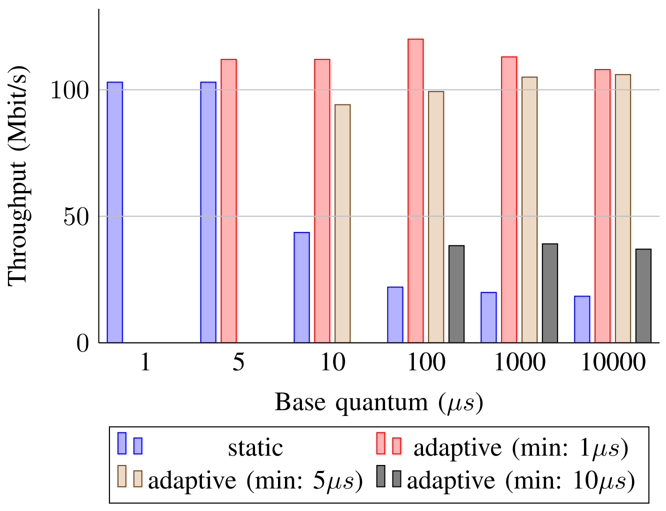
TABLE I: Annotation candidates for iperf3 benchmark

The Linux driver for ethoc registers an interrupt handler. In case a packet is sent or received, the interrupt handler is disabled and the network device is switched to polling mode. This is done to reduce interrupt overhead under the assumption that additional packet transfers will follow. Once no more packets are in ethoc's send buffer, the driver leaves polling mode and reenables the interrupt handler.

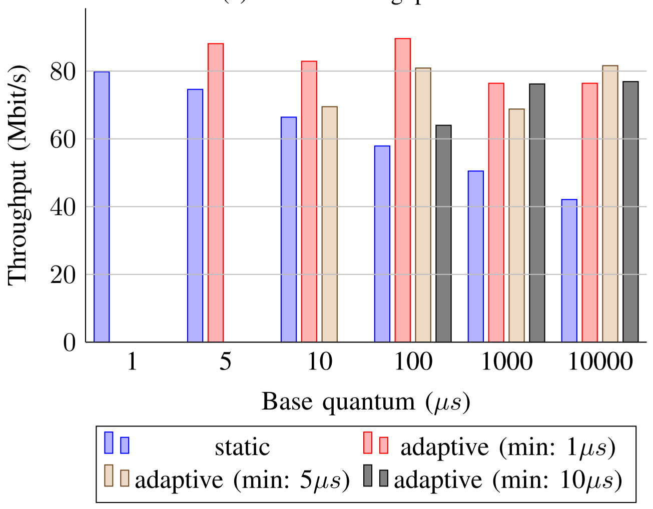
To optimize the performance of iperf3 on AVP64, a trace is recorded during benchmark execution. The resulting trace file is analyzed using the trace analysis tool to yield candidate functions with their corresponding transaction and interrupt activity, which are shown in Table I.

The IRQ factor describes how often a function is executed in quanta with a pending IRQ, compared to quanta without one, while the transactions indicate the amount of induced

[1]https://github.com/aut0/avp64
[2]https://fasterdata.es.net/performance-testing/network-troubleshooting-tools/iperf/



(a) Receive-throughput



(b) Send-throughput

Fig. 6: Receive and send throughput for different temporal decoupling schemes

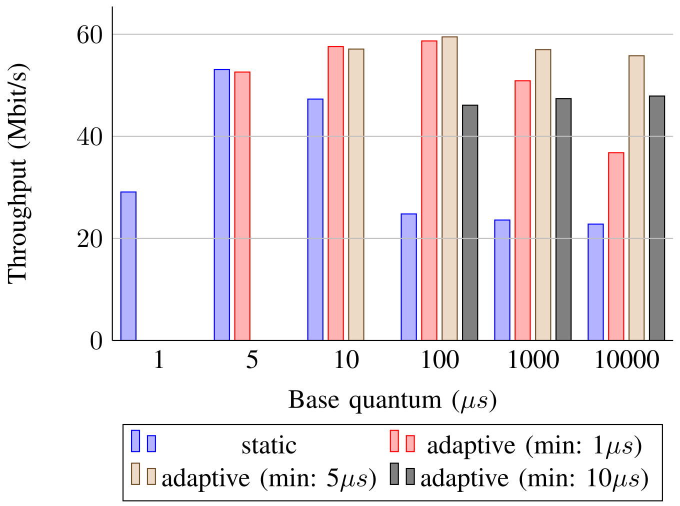
MMIO. The `ethoc_interrupt` and the `ethoc_poll` functions are part of the ethoc driver. The first one is called when an ethoc interrupt is received, while the second one is called after the device is switched to polling mode. Both can be annotated for the adaptive quantum approach. The input parameters were chosen as $A = 0.5, B = 0.1, C = 1\,\mu s$. The results are presented in Fig. 6 for receiving and sending data. The adaptive quantum scheme with different q_{min} and q_{base} is compared to a static quantum with the size of q_{base}. Both the receive and send throughput profit from the adaptive quantum. For the static quantum the TCP throughput is reduced with rising quantum size. In the receive scenario, performance is significantly reduced for quantum sizes larger than 5 µs, because the incoming packets cannot be processed fast enough by the ethoc model since more simulation time is spent in the CPU model. Receive performance is constant above 100 µs, because the incoming packets are processed in one quantum and the CPU executes a Wait-for-Interrupt instruction which ends the quantum early effectively reducing quantum size.

The adaptive quantum is beneficial to the TCP throughput for all tested minimum quantum durations and in both the send and receive scenario. It should be noted, that both considered annotation candidates produce the same result. In the receive scenario, the adaptive quantum with a q_{min} of 1 µs performs best. It leads to a performance improvement between 1.08x and 5.87x over the static quantum depending on the base quantum duration. In the send scenario, the adaptive quantum with a q_{min} of 1 µs performs best in most cases except for the largest tested base quantum duration of 10 000 µs. A performance improvement of between 1.18x and 1.81x is achieved. In the latter case, the adaptive quantum with a q_{min} of 5 µs performs better, reaching 1.94x of speedup.

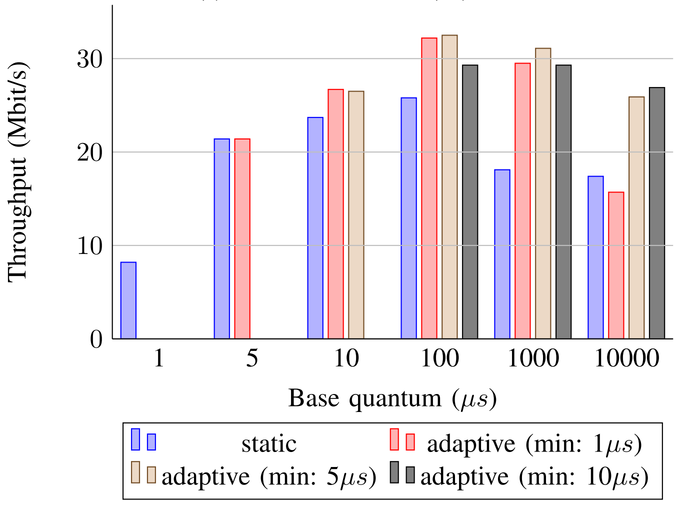
As the quantum is effectively reduced with the adaptive quantum approach, it could be speculated that even though throughput is increased, the compute performance of the VP as a whole is reduced. A second benchmark with additional compute load for the CPU model is conducted to investigate this effect. For this, data is again transferred between host and VP via the virtual ethoc device and the throughput is measured. This time the SHA256 hash of the received data is computed in the VP. To simulate higher compute load, the hash function is applied multiple times. The results are shown in Fig. 7. Again a static quantum approach is compared to our adaptive quantum with different minimum quantum size q_{min}. The input parameters were left unchanged. It can be observed that for increasing amounts of computation the static quantum sweet spot moves towards higher quantum durations, as the increased compute load profits from them.

However, our adaptive quantum approach is still able to outperform the static quantum in most cases. Compared to the previous benchmark, the ideal q_{min} for the adaptive approach is at 5 µs instead of 1 µs. This is plausible as the added compute load requires a larger quantum duration to be calculated efficiently. When the SHA256 hash of the received data is calculated once, the static quantum performs best with a base quantum duration of 5 µs. The adaptive quantum is not able to increase performance at this base quantum duration. This is because reducing the quantum size decreases performance for the compute task. For larger base quanta the adaptive quantum with a q_{min} of 5 µs outperforms the static quantum by a factor between 1.21x and 2.45x. Only with a base quantum of 10 µs a q_{min} of 1 µs performs slightly better.

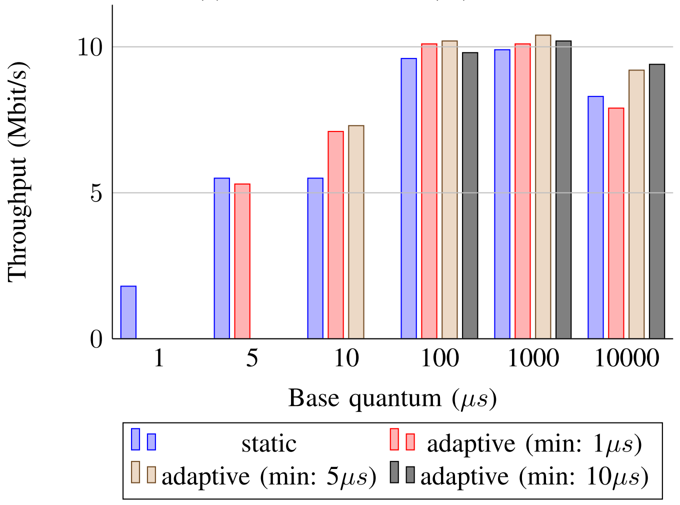
A similar picture is painted for the increased load scenarios where the SHA256 is calculated 5 or 25 times. In the first case, the adaptive quantum with a q_{min} of 5 µs performs best in most scenarios. Speedups between 1.21x and 1.72x over the static quantum are achieved. The overall throughput is reduced, as the CPU model requires time to calculate the hash function. At a base quantum of 5 µs the adaptive approach is not able to outperform the static quantum, but performs on par. For the second case, the throughput is further reduced, but the compute intensity of the task has been significantly increased. The adaptive quantum with a q_{min} of 5 µs performs best and is able to achieve speedups between 1.05x and 1.33x over the static quantum. At a base quantum duration of 10 000 µs the



(a) Receive + SHA256 (x1)



(b) Receive + SHA256 (x5)



(c) Receive + SHA256 (x25)

Fig. 7: Receive throughput with varying SHA256 compute load

adaptive quantum with a q_{min} of 10 µs performs slightly better achieving a speedup of 1.13x instead of 1.11x. However, at a base quantum duration of 5 µs the adaptive quantum performs slightly worse than the static quantum.

V. Conclusion

In this work, a novel approach to temporal decoupling in SystemC TLM 2.0 Virtual Platforms (VPs) was presented. Our method differentiates itself by taking the behavior and requirements of the target software into account when setting the SystemC quantum duration during simulation. The quantum duration is adapted to the target software to optimize the simulation performance. This is achieved in a two step process. First the target software behavior is analyzed during a profiling run. Interactions between target software and VP are captured and stored to a trace file. The trace is used as the basis for the analysis. Using our analysis tool, annotation points in the target software are determined. These annotation points are exported to a candidate function file that is used for annotation in the VP. In the second step of the process, the annotation points are used by the VP during runtime to adapt the quantum size to the target software behavior. The achievable performance gains were shown in a representative case study with speedups between 1.08x and 5.87x.

For future work we will test our method on more complex benchmarks, as finding good annotation candidate functions is not trivial. Adding Linux kernel instrumentation to trace user space process execution when hardware is accessed from user space, e.g. via Userspace IO drivers, would extend the applicability of our method. Furthermore, adding instrumentation to trace interaction between different cores in a multi-core system is an option. The method relies on the availability of debug information. It could be extended to also cover stripped binaries without any additional information.

Acknowledgment

We thank Audi AG for supporting this work.

References

[1] "Standard systemc LRM," *IEEE Std 1666-2011*, 2012.
[2] Damm et al., "Connecting systemc-ams models with OSCI TLM 2.0 models using temporal decoupling," in *FDL*. IEEE, 2008.
[3] Gläser et al., "Temporal decoupling with error-bounded predictive quantum control," in *FDL*. IEEE, 2015.
[4] Jünger et al., "Optimizing temporal decoupling using event relevance," in *ASPDAC*. ACM, 2021.
[5] Jung et al., "Speculative temporal decoupling using fork()," in *DATE*. IEEE, 2019.
[6] Weinstock et al., "Time-decoupled parallel systemc simulation," in *DATE*. IEEE, 2014.
[7] ——, "Systemc-link: Parallel systemc simulation using time-decoupled segments," in *DATE*. IEEE, 2016.
[8] Stattelmann et al., "Fast and accurate source-level simulation of software timing considering complex code optimizations," in *DAC*. ACM, 2011.
[9] Razaghi el al., "Host-compiled multicore system simulation for early real-time performance evaluation," *ACM Trans. Embed. Comput. Syst.*, vol. 13, no. 5s, pp. 166:1–166:26, 2014.
[10] Bellard, "Qemu, a fast and portable dynamic translator," in *USENIX*. USENIX, 2005.