Towards High-Performance Virtual Platforms: A Parallelization Strategy for SystemC TLM-2.0 CPU Models

Nils Bosbach
RWTH Aachen University
Aachen, Germany

Niko Zurstraßen
RWTH Aachen University
Aachen, Germany

Rebecca Pelke
RWTH Aachen University
Aachen, Germany

Lukas Jünger
MachineWare GmbH
Aachen, Germany

Jan Henrik Weinstock
MachineWare GmbH
Aachen, Germany

Rainer Leupers
RWTH Aachen University
Aachen, Germany

ABSTRACT

SystemC TLM-2.0 is currently the industry standard for simulating full Systems-on-a-Chip (SoCs). Although SystemC is designed to simulate the behavior of complex, parallel systems, the simulation itself is by default single-threaded. We present a technique to overcome this performance limitation by parallelizing the CPU model of a SystemC-TLM-2.0-based system-level simulator, a so-called Virtual Platform (VP). Our solution is fully compliant with the SystemC standard. To further increase the performance, we developed algorithms for asynchronous DMI pointer caching and we introduced a new tunable parameter called `async_rate`. This parameter controls the frequency used to annotate timing information to SystemC.

Evaluation results demonstrate a significant speedup compared to sequential execution, with a maximum of 7.8 x achieved for octa-core VPs on fully parallelizable workloads. For the execution of the NPB suite on the SIM-V VP, an average speedup of 6.2 x is achieved. This approach is a promising solution for accelerating VPs while adhering to the SystemC standard.

CCS CONCEPTS

• **Hardware → Hardware-software codesign**; • **Computing methodologies → Discrete-event simulation**.

KEYWORDS

SystemC, TLM, CPU, VCML, sc-during, parallel

ACM Reference Format:
Nils Bosbach, Niko Zurstraßen, Rebecca Pelke, Lukas Jünger, Jan Henrik Weinstock, and Rainer Leupers. 2024. Towards High-Performance Virtual Platforms: A Parallelization Strategy for SystemC TLM-2.0 CPU Models. In *61st ACM/IEEE Design Automation Conference (DAC '24), June 23–27, 2024, San Francisco, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3649329.3658257

DAC '24, June 23–27, 2024, San Francisco, CA, USA

ACM ISBN 979-8-4007-0601-1/24/06…$15.00
https://doi.org/10.1145/3649329.3658257

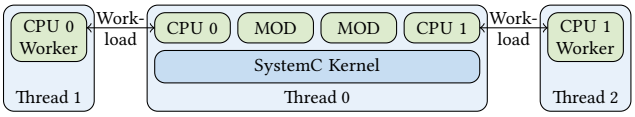


Figure 1: Parallelization scheme.

1 INTRODUCTION

In the ever-evolving landscape of embedded system design and validation, Virtual Platforms (VPs) have become essential tools in the industry. VPs simulate hardware systems in software, enabling the testing and execution of unmodified target software even before physical hardware is available. By parallelizing the hardware and software development phases, VPs significantly improve the design process's efficiency, thereby reducing the time-to-market.

The industry standard for VP development is SystemC with its Transaction-Level Modeling (TLM)-2.0 extension [9]. TLM-2.0 provides an abstract communication interface between diverse models. Through standardization, SystemC-based models can easily integrate into a wide range of simulations. A drawback of SystemC is that the simulation runs in a single thread. The Discrete Event Simulation (DES) scheduler of SystemC processes events in sequence, limiting the performance gains modern x86 workstations and comparable parallel host hardware offer.

To test large target-software stacks and perform multiple test cycles, high performance of VPs is crucial. In recent years, many different approaches have been developed to parallelize SystemC. Most of them require manual adjustments such as partitioning or the use of custom functions which reduces the usability.

To avoid constraining other VP models, we present an approach that aims at parallelizing the CPU model instead of the entire simulation at once. Since the CPU model is typically the most computation-heavy model of a VP, parallelizing this model accelerates the entire simulation. As shown in Fig. 1, each simulated CPU core gets its asynchronous thread that runs the Instruction-Set Simulator (ISS). When a CPU needs to interact with other models, the communication is pushed back to the main thread to remain SystemC compliant. We present the following contributions:

- **CPU model parallelization**: We introduce a standard-compliant algorithm to parallelize a SystemC-TLM-based CPU model.
- **Ensuring thread safety**: For non-thread-safe ISSs, we developed the *isolation guard*, a deadlock-free synchronization mechanism.
- **Optimization of the timing-annotation frequency**: Our introduced parameter `async_rate` steers the frequency the CPU model communicates timing annotations to the SystemC kernel.

PREPRINT - accepted by the 61st ACM/IEEE Design Automation Conference (DAC '24) - DOI: 10.1145/3649329.3658257

- **Memory-access optimization**: Asynchronous DMI pointer caching enables direct and fast memory access from a CPU thread.
- **Performance evaluation**: We show substantial improvements of up to 7.8 x for octa-core setups of two different VPs.

2 BACKGROUND AND RELATED WORK

SystemC [9] is the current standard for Electronic System Level (ESL) simulation. It supports different levels of abstraction. Especially in the early days, Register-Transfer Level (RTL)-like simulations using `sc_signals` were widely used. During this time, parallelization approaches mainly focused on delta-cycle parallelization [6–8, 14, 18, 20–22, 24, 31]. A delta cycle is an internal update step within a time step. Table 1 gives an overview of the different works and shows their parallelization technique. Since `sc_signal`-based simulation is overly detailed and therefore not suitable for high-performance full-system simulation, we do not provide an in-depth analysis of delta-cycle-based parallelization approaches.

To achieve higher performance, the level of abstraction can be increased. When the use of `sc_signals` is omitted, delta cycles are eliminated. Instead, modules are connected using the more abstract TLM-2.0 sockets. Temporal decoupling [9] can be applied, allowing modules to run ahead of the simulation time. This technique increases the performance by reducing the number of synchronizations. The downside is reduced timing accuracy. To manage this tradeoff between performance and timing accuracy, the *quantum* parameter can be used. It defines the maximum amount of simulation time a module is allowed to run ahead before synchronizing.

In 2015, Becker et al. showed that delta-cycle-based parallelization approaches hardly provide any performance improvement for temporally decoupled VPs [3]. Therefore, new parallelization approaches have been developed to execute full quanta of different models in parallel [4, 16, 17, 19, 23, 28–30]. Various locations where parallelization can be implemented exist, ranging from the SystemC kernel itself to supplementary libraries, the compiler, and specialized hardware. A common approach is to create a custom SystemC kernel with parallelization capabilities [4, 16, 23]. These approaches distribute the execution of `SC_THREAD`s and `SC_METHOD`s across multiple threads. The drawback is that mapping `SC_THREAD`s and `SC_METHOD`s to different workers is a complex task that has an enormous impact on performance. Therefore, manual handling is typically necessary, which decreases usability.

Other approaches use the standard kernel but divide the simulation into several parts that run in parallel [19, 28–30]. Each part has its own sequential scheduler. The challenge is to synchronize the simulation time and the communication between the segments.

Table 1: SystemC parallelization approaches.

Work	Parallelization	Location	SystemC Compliant
[6–8, 20, 21]	Delta cycles	Custom kernel	⊗
[31]	Delta cycles	Co-simulation	⊗
[22]	Delta cycles	Custom kernel, special HW	✓
[14]	Delta cycles	Custom compiler, simulator	⊗
[18, 24]	Delta cycles	GPU execution	⊗
[4, 16, 23]	Quantum	Custom kernel	✓
[19, 30]	Quantum	Co-simulation	⊗
[28, 29]	Quantum	Connected segments	⊗
[17]	Quantum	Additional library	✓
This work	Quantum	Parallel CPU model	✓

In 2013, Moy introduced *sc-during* [17]. He provides an additional library, sc-during, that can be used to offload a task to another thread and annotate the timing of the offloaded task. It does not automatically parallelize a pre-existing simulation but instead enables the manual parallelization of specific parts and components.

Most approaches that aim to parallelize an entire simulation at once have the problem that they impose requirements on other models. For example, if the SystemC kernel is replaced by a non-compliant kernel with parallelization capabilities, existing models must be adapted to work with that specific kernel. After the adaptation, they are no longer standard-compliant, i.e., they do not work with other SystemC simulations. For this reason, our approach focuses on the CPU model. We selected the CPU model due to its ubiquitous presence in VPs and typical high computational demands. By using sc-during as the parallelization technique, the model remains standard-compliant.

Our parallelization method is integrated into the open-source Virtual Components Modeling Library (VCML) [27]. VCML expands SystemC's functionality by including widely used models, TLM-2.0-based communication protocols, and fundamental building blocks. Additionally, VCML incorporates a processor class that can encapsulate an ISS into the SystemC framework. Several VP use the VCML processor class with an embedded ISS as their CPU model. Examples of such VPs are SIM-V [12] (RISC-V), AVP64 [10] (ARM), and OR1KMVP [26] (OpenRISC 1000). AVP64 and OR1KMVP are fully open-source. This paper aims to enhance the processor class by incorporating parallelization support. By employing this approach, all VPs that are using the processor class to incorporate an ISS experience the advantages of parallelization.

3 PARALLELIZATION APPROACH

As a starting point for our parallelization approach, we use VCML's processor class. The aim is to offload the execution of the ISS to a second thread. For this, sc-during is used. VCML has an internal implementation of sc-during. It provides the following functions that we utilize to parallelize VCML's processor model:

- `sc_async`: This method runs a designated function, which is passed as a parameter, in a separate thread.
- `sc_sync`: A function, which is given as a parameter, is executed within the main thread.
- `sc_progress`: This function can be called from an asynchronous thread to allow the simulation in the main thread to advance for a specified time interval.

3.1 Parallel CPU Model

The processor class includes a virtual `simulate` function utilized for running the ISS during simulation. This function is executed numerous times during simulation by an `SC_THREAD` to operate the ISS and simulate a specific number of instructions (a quantum). When the function returns, the elapsed simulation time is annotated to SystemC. Instead of direct execution within the main thread, the processor class is modified to utilize `sc_async` for invoking the `simulate` function. This non-blocking function offloads the ISS execution to another thread, thereby allowing the simulation to proceed in the main thread. Although this implementation creates a preliminary parallel setup, it faces two issues:

PREPRINT - accepted by the 61st ACM/IEEE Design Automation Conference (DAC '24) - DOI: 10.1145/3649329.3658257

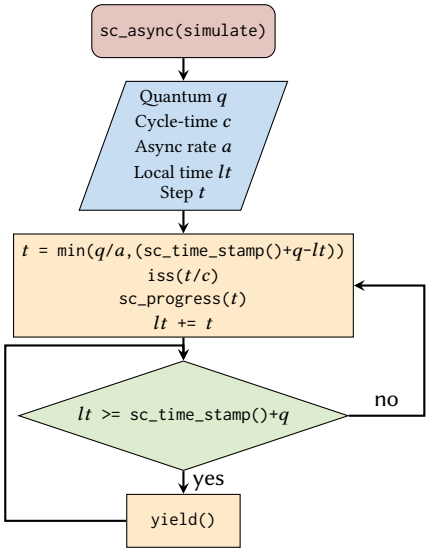


Figure 2: Asynchronous simulate loop.

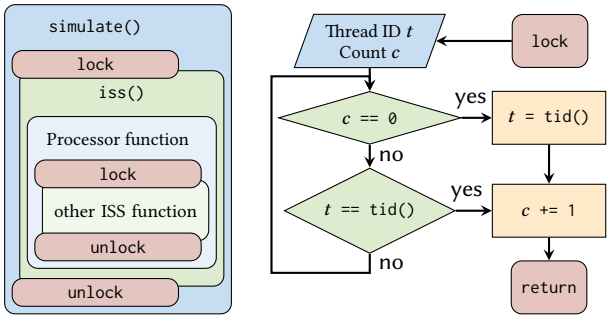
(1) When the ISS interacts with other models, it performs the interaction in the asynchronous thread, which is non-compliant with the SystemC standard. For instance, the `b_transport` function utilized for TLM-based communication is required to be invoked from the main thread.
(2) The simulation time increases only after all ISSs have completed simulating their specified number of instructions, i.e., when the full quantum is exceeded. This creates a synchronization barrier that results in reduced performance.

To address the first concern, it is important to ensure that the communication between the CPU model and other models is carried out in the main thread. This communication takes place, e.g., when the CPU model accesses the memory or peripherals. A typical example of this is the execution of load and store instructions in the ISS. To gain access to the memory, the ISS calls the `b_transport` function of the processor's TLM-2.0 socket. Since the ISS runs in the asynchronous thread, the `sc_sync` function must be utilized to redirect the call back to the main thread. This guarantees that other models are always accessed from the main thread and enables the use of SystemC's `wait` function within `b_transport` calls. Please note that for instruction fetching, the used CPU models do Direct Memory Interface (DMI) accesses from the asynchronous thread.

The second significant challenge arises from synchronization barriers in our approach. Simulation time progresses only when all CPU cores complete a full quantum. To address this, we introduce interruptions in the ISS execution within each quantum. There, we annotate the simulation progress of the ISS to the SystemC kernel using the `sc_progress` function to allow the simulation to proceed. The number of interruptions per quantum is controlled by the `async_rate`, a new optimizable parameter.

Our algorithm for calculating the number of instructions the ISS should simulate during a call without interruption is illustrated in Fig. 2. It uses the following variables:

- q: Quantum length for temporal decoupling.
- c: Time per instruction cycle.
- a: Asynchronous interruption rate.



(a) Deadlock example. **(b) The `lock` function.**

Figure 3: Isolation guard.

- lt: Local SystemC timestamp.
- t: ISS step size.

The algorithm computes the step size t as part of the quantum q/a unless it surpasses a full quantum's time. Afterward, the ISS simulates t/c instructions and communicates the interval to the SystemC kernel via `sc_progress` before updating lt. If the process ran ahead of simulation time for a full quantum, it waits for the other processes to catch up. Otherwise, it proceeds.

3.2 Synchronization Primitives

In our approach, the CPU model only accesses other models from the main thread to avoid the need for thread safety in those models. However, problems may occur if the embedded ISS lacks thread safety. Let us imagine a scenario where the interrupt controller signals an interrupt to the CPU model while the ISS is active. This interrupt comes from the main thread, which primarily runs the simulation. If the ISS is active, two ISS functions run concurrently in separate threads. This could result in issues if the ISS implementation is not thread-safe, potentially leading to erroneous behavior.

To address this issue, we suggest implementing an *isolation guard* with a locking mechanism before calling an ISS function and an unlocking mechanism after the call. The interface of the isolation guard resembles the one of a conventional mutex, but its functionality differs slightly. Using a mutex to protect ISS accesses can result in deadlocks. A common scenario illustrating the occurrence of deadlocks when using a traditional mutex is illustrated in Fig. 3a. It sketches nested calls of ISS functions with functions of the VCML processor class in between. When the processor class calls a function of the ISS, the call needs to be protected by surrounded `lock`/`unlock` calls of the mutex. During ISS simulation, certain instructions require a call to the processor class to broadcast information to all cores of the VP, including its own. An additional call from the processor model to its ISS, surrounded by `lock`/`unlock`, will fail due to the lock maintained by the `simulate` function.

To overcome this problem, our isolation guard allows nested lock calls within the same thread, while blocking concurrent accesses from different threads. Fig. 3b illustrates our lock mechanism, which uses two variables. The thread ID t serves to identify the current lock holder and the counter c tracks nested lock calls. At first, the isolation guard checks if any thread is holding the lock. If not, it stores the current thread's ID in t, increments c, and returns. If the counter c exceeds zero, the lock function checks whether the lock belongs to the current thread. If so, it increments c and grants a

PREPRINT - accepted by the 61st ACM/IEEE Design Automation Conference (DAC '24) - DOI: 10.1145/3649329.3658257

successful call. Otherwise, if the lock is assigned to another thread, the current thread must wait until the lock-holding thread releases all locks using the unlock function. The `unlock` function simply decrements the counter c.

Please note that Fig. 3b provides a basic outline of the lock algorithm, and the actual implementation uses a mutex to synchronize access to t and c. Additionally, optimization is applied to handle cases where another thread obtains the lock using condition variables for efficient waiting.

3.3 Asynchronous DMI Caching

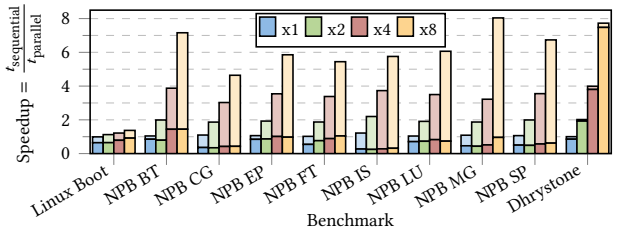
In the current implementation, processor calls that access other models are transferred from the asynchronous thread to the main thread to comply with the standard, resulting in sequentialized memory accesses. This sequentialization has a significant impact on overall performance, especially for memory-intensive workloads. When the processor model accesses memory, it sends a TLM-2.0 transaction to the memory model. The model can provide a DMI pointer for further memory access without the need for additional transactions. VCML's TLM-2.0 socket of the processor model caches the DMI pointers, but the cache is accessed only after the read or write operation is transferred back to the main thread.

To enhance memory accesses, we suggest a method that first checks the DMI cache from the asynchronous thread before sending a TLM-2.0 transaction. If the requested address is present in the DMI cache (a cache hit), direct memory access is performed from the asynchronous thread. If there is a cache miss, e.g., because the address corresponds to a peripheral that does not support DMI, a TLM-2.0 transaction is sent from the main thread. This method enables DMI accesses from an asynchronous thread without the added expense of transferring requests to the sequential main thread.

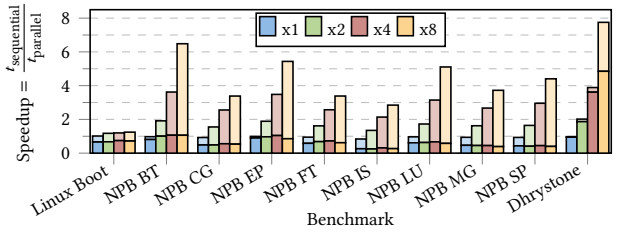
4 EVALUATION

To assess the efficacy of our parallelization method, we implemented the asynchronous simulate loop (as shown in Fig. 2) in VCML's processor class. We then evaluated the resulting performance improvements using two VPs developed within the VCML framework. The first VP is constructed upon SIM-V [15], a RISC-V simulator. It mimics a minimalistic system that includes a 64-bit RISC-V CPU with a configurable number of Hardware Threads (harts), a Platform-Level Interrupt Controller (PLIC), a Core-Local Interrupt Controller (CLINT), main memory, and a Universal Asynchronous Receiver/Transmitter (UART) device for interaction and control. The second VP, the open-source ARMv8 Virtual Platform (AVP64) [13], features a comparable architecture to SIM-V but with an ARM-based CPU. It is composed of ARM-based CPU cores, a Generic Interrupt Controller (GIC), main memory, and a UART interface. To synchronize accesses to the non-thread-safe ISS of AVP64, an isolation guard is used for this VP as described in Section 3.2.

Our performance evaluation includes several benchmarks. First, we analyze the boot process of a Buildroot Linux Operating System (OS). Next, we utilize the NAS Parallel Benchmarks (NPB) [2] suite based on Open Multi-Processing (OpenMP) [5] to evaluate performance under various computational workloads. As a third



(a) SIM-V.



(b) AVP64.

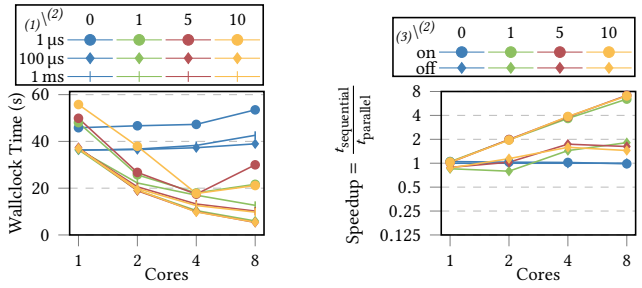
Figure 4: Parallelization speedup. The lower part of a bar shows the worst-case speedup. The full bar represents the best-case speedup. Quantum: 100 µs.

benchmark, we run a bare-metal Dhrystone [25], executed individually for each CPU core. This benchmark is chosen for its optimal parallelizability, as it includes minimal inter-core communication and synchronization. Our analysis explores the effect of varied simulation parameters, including quantum, `async_rate`, and `async_dmi`. We conducted all benchmarks on an *AMD Ryzen 9 3900X* processor.

Fig. 4 displays the possible speedup results that can be obtained through our parallelization method for various benchmarks on the x-axis. The graph uses different colors to distinguish the number of simulated CPU cores. To compute the speedup, one must divide the execution time of the sequential execution (with `async_rate` = 0 and `async_dmi` = off) by the execution time of the parallelized version. As performance is dependent on the values of `async_rate` and `async_dmi`, Fig. 4 demonstrates the minimum and maximum speedup values for all tested configurations. The bottom sections of the bars depict the minimum attainable speedup, which corresponds to the least favorable parameter combinations of `async_rate` $\in \{1, 5, 10\}$ and `async_dmi` $\in \{\text{on}, \text{off}\}$. The full bars represent the speedup achieved with the most favorable parameters. The selection of the optimal parameter will be discussed in more detail.

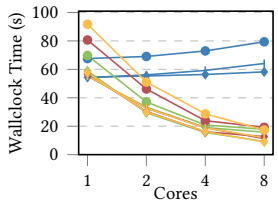
For the execution of the NPB suite, the octa-core VPs achieve an average speedup of 6.2 x for SIM-V and 4.3 x for AVP64. The lower speedup of AVP64 is caused by the requirement of an isolation guard for ensuring thread safety of the ISS. In the context of the Dhrystone benchmark, the octa-core configurations of both VPs achieve a maximum speedup of 7.8 x. This equates to a utilization of 97.5 % of the maximum potential speedup value of 8 x for perfectly parallelizable workloads. It should be noted that while the computational aspects of the benchmark are entirely parallelizable, there exist synchronized sections in the setup and teardown phases that marginally decrease the potential for parallelization.

PREPRINT - accepted by the 61st ACM/IEEE Design Automation Conference (DAC '24) - DOI: 10.1145/3649329.3658257

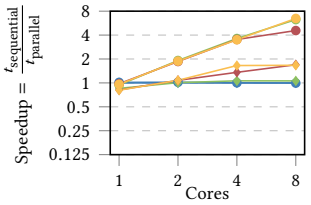


(a) SIM-V execution time. *(3)*: on.

(b) SIM-V speedup. *(1)*: 100 µs.

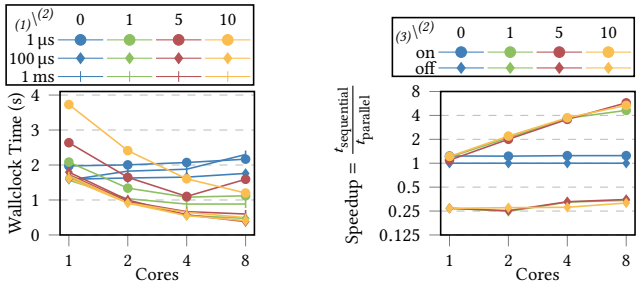


(c) AVP64 execution time. *(3)*: on.



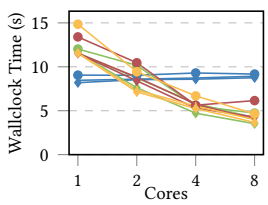
(d) AVP64 speedup. *(1)*: 100 µs.

Figure 5: NPB BT benchmark execution results. Legend: *(1)* quantum, *(2)* `async_rate`, *(3)* `async_dmi`

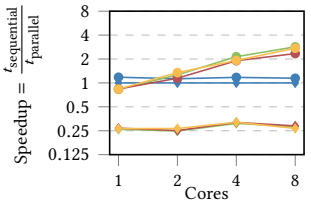


(a) SIM-V execution time. *(3)*: on.

(b) SIM-V speedup. *(1)*: 100 µs.



(c) AVP64 execution time. *(3)*: on.



(d) AVP64 speedup. *(1)*: 100 µs.

Figure 6: NPB IS benchmark execution results. Legend: *(1)* quantum, *(2)* `async_rate`, *(3)* `async_dmi`

For predominantly sequential workloads, such as the Linux boot process, Amdahl's law [1] specifies that speedup gains from parallelization are restricted. Consequently, the speedup achieved for the Linux boot is approximately 1. Occasionally, incorporating parallelization approaches can generate performance impediments when simulating non-parallelizable workloads. This results in a speedup of less than one. For instance, this is true for the NPB IS benchmark's lowest achieved speedup values. The benchmark's worst parameter configuration causes speedups of only 0.25 x. Further analysis will highlight the specific cases where these speed reductions occur and recommend prevention measures. However, if one sets the parameters wisely, the speedup is always larger than one.

The range between the highest and lowest speedup values, as depicted in Fig. 4, highlights the impact of the parameters `async_rate` and `async_dmi` on overall performance. SIM-V generally achieves higher speedups compared to AVP64 because it is tread-safe and therefore does not require an isolation guard. This guard can delay the execution of a CPU core or even the whole simulation. For example, when an interrupt is triggered during core simulation, it must be waited for the ISS to complete its simulation before signaling the interrupt, leading to delays and reduced performance.

To gain a deeper understanding of how the `async_rate` and `async_dmi` parameters affect performance, we present the results of the NPB BT benchmark for SIM-V and AVP64 in Fig. 5. The benchmark implements a block tri-diagonal solver for a system of partial differential equations [2]. Out of the NPB benchmarks, it achieves one of the highest speedups after parallelization for both VPs. Figs. 5a and 5c illustrate the benchmark's execution time for varying numbers of simulated cores, different quanta, and different `async_rate` parameters while keeping `async_dmi` activated. During sequential execution (represented by blue lines, `async_rate` = 0), the wallclock time experiences a slight increase as the number of simulated cores increases. This is due to the overhead generated by distribution and synchronization. For parallelized execution, the needed wallclock time is reduced for an increased amount of cores.

It is noticeable that the influence of the `async_rate` parameter is comparable to the one of the quantum parameter as shown in Figs. 5a and 5c. Together, these parameters determine the number of cycles that the ISS simulates in a single call. The key differentiation is that, following an interruption from `async_rate`, the simulation-time progress is annotated to the SystemC kernel without blocking. At the end of a quantum, in contrast, the ISS's execution must be halted until the rest of the simulation catches up.

While multiple combinations of quantum and `async_rate` may result in the same number of instructions the ISS needs to simulate in a single call, their performance may vary. For example, the combination of a 100 µs quantum and an `async_rate` of 1 (refer to Figs. 5a and 5c, diamond shape markers on green line) results in a 100 µs simulation interval for the ISS. The identical interval may be accomplished by utilizing a 1 ms quantum and an `async_rate` of 10 (see Figs. 5a and 5c, orangish line with straight indexes). The 100 µs quantum and `async_rate` of 1 results in marginally superior performance under these circumstances.

To select the optimal value for the `async_rate` parameter, we suggest following a similar approach to selecting the quantum value. The most straightforward method is to evaluate different values and select the parameter that yields the best performance for the workload. More advanced techniques, such as adaption during simulation [11] or analytical evaluations [32], are also possible.

Figs. 5b and 5d illustrate the speedups attained for various combindations of `async_rate` and `async_dmi`, using a quantum of 100 µs. It is noteworthy that when the `async_dmi` parameter is disabled, the performance gains are restricted. This constraint occurs because, in such cases, all communication between the CPU and other models is serialized by transferring the workload back to the

main thread. Therefore, it is clear that the activation of `async_dmi` is crucial for achieving significantly improved performance values.

Fig. 6 illustrates the performance values of the NPB IS benchmark, which applies the memory-intensive bucket sort algorithm to integers [2]. Compared to the BT benchmark, shown in Fig. 5, which demonstrates a workload that attains high speedup values when parallelized, the IS benchmark is an example of a workload with a lower speedup. In Figs. 6a and 6c, it is evident that the simulation becomes more sensitive to the chosen `async_rate` as the number of cores increases. This increased sensitivity is due to a larger number of threads, where the synchronization frequency significantly affects performance. Furthermore, Figs. 6b and 6d emphasize the crucial significance of `async_dmi` for memory-intensive workloads. Disabling `async_dmi` results in a staggering 75 % performance drop compared to the sequential execution. This decrease can be attributed to a large number of memory accesses during the benchmark, which causes a significant portion of the simulation to return from the asynchronous thread to the main thread. As a result, the simulation reverts to a sequential execution mode, accompanied by increased overhead due to the involvement of multiple threads and their associated synchronization requirements. Enabling `async_dmi` proves to be instrumental in preventing slowdowns and facilitating fast, parallel DMI accesses, ultimately enhancing the overall performance of the simulation.

5 CONCLUSION AND FUTURE WORK

In this paper, we present a robust parallelization technique for SystemC-TLM-2.0-based VPs that significantly enhances simulation performance. Our approach optimizes CPU-model execution within VPs by utilizing the sc-during method and asynchronous DMI pointer caching, while still ensuring adherence to the SystemC standard. We introduce the `async_rate` parameter, providing fine-grained control over the simulation's timing-annotation frequency. The implementation of our isolation guard guarantees thread safety for embedded ISSs that do not provide those capabilities. By implementing the approach in the processor base class of the modeling library VCML, all VCML-based VPs profit from parallelization.

Our evaluation presents noteworthy speedup outcomes, achieving a maximum speedup of 7.8 x on an octa-core VP for highly parallelizable workloads. For the NPB suite executed in a Linux environment, an average speedup of 6.2 x for the RISC-V VP SIM-V can be reached. These results demonstrate the significant performance advantages of our approach, indicating the potential for expediting the simulation of intricate VPs.

As a next step, the creation of automated parameter tuning mechanisms that can dynamically adjust simulation parameters such as `async_rate` and `async_dmi` could heighten the adaptability and effectiveness of our parallelization approach. This would guarantee top-notch performance across a variety of workloads.

REFERENCES

[1] Gene M. Amdahl. 1967. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference (AFIPS '67 (Spring))*.

[2] David Bailey, Tim Harris, William Saphir, Rob Van Der Wijngaart, Alex Woo, and Maurice Yarrow. 1995. *The NAS parallel benchmarks 2.0.* Technical Report.

[3] Denis Becker, Matthieu Moy, and Jerome Cornet. 2015. Challenges for the parallelization of loosely timed SystemC programs. In *2015 RSP*.

[4] Gabriel Busnot, Tanguy Sassolas, Nicolas Ventroux, and Matthieu Moy. 2021. Standard-compliant parallel SystemC simulation of loosely-timed transaction level models: From baremetal to Linux-based applications support. *Integration* 79 (July 2021).

[5] Rohit Chandra, Leo Dagum, David Kohr, Ramesh Menon, Dror Maydan, and Jeff McDonald. 2001. *Parallel programming in OpenMP.* Morgan kaufmann.

[6] Bastien Chopard, Philippe Combes, and Julien Zory. 2006. A conservative approach to systemc parallelization. In *Proceedings of the 6th ICCS*. Springer-Verlag.

[7] Moo-Kyoung Chung, Jun-Kyoung Kim, and Soojung Ryu. 2014. SimParallel: A high performance parallel SystemC simulator using hierarchical multi-threading. In *2014 IEEE International Symposium on Circuits and Systems (ISCAS)*. 1472–1475.

[8] Philippe Combes, Eddy Caron, Frédéric Desprez, Bastien Chopard, and Julien Zory. 2008. Relaxing Synchronization in a Parallel SystemC Kernel. In *2008 IEEE International Symposium on Parallel and Distributed Processing with Applications*.

[9] IEEE Standards Association and others. 2012. IEEE Standard for Standard SystemC Language Reference Manual. *IEEE Std 1666-2011* (Jan. 2012).

[10] Lukas Jünger. 2023. AVP64. https://github.com/aut0/avp64

[11] Lukas Jünger, Carmine Bianco, Kristof Niederholtmeyer, Dietmar Petras, and Rainer Leupers. 2021. Optimizing Temporal Decoupling using Event Relevance. In *Proceedings of the 26th Asia and South Pacific Design Automation Conference*.

[12] Lukas Jünger, Jan Henrik Weinstock, and Rainer Leupers. 2022. SIM-V: Fast, Parallel RISC-V Simulation for Rapid Software Verification. In *Proceedings of DVCon Europe 2022*. Munich.

[13] Lukas Jünger, Jan Henrik Weinstock, Rainer Leupers, and Gerd Ascheid. 2019. Fast SystemC Processor Models with Unicorn. In *Proceedings of the Rapid Simulation and Performance Evaluation: Methods and Tools (RAPIDO '19)*. ACM.

[14] Guantao Liu, Tim Schmidt, Zhongqi Cheng, Daniel Mendoza, and Rainer Dömer. 2019. RISC Compiler and Simulator, Release V0.6.0: Out-of-Order Parallel Simulatable SystemC Subset. *Technical Report CECS-TR-19-04, Center for Embedded and Cyber-physical Systems, University of California, Irvine* (2019).

[15] MachineWare. 2023. MachineWare. https://www.machineware.de/

[16] Aline Mello, Isaac Maia, Alain Greiner, and Francois Pecheux. 2010. Parallel simulation of systemC TLM 2.0 compliant MPSoC on SMP workstations. In *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*.

[17] Matthieu Moy. 2013. Parallel programming with SystemC for loosely timed models: A non-intrusive approach. In *2013 DATE*. IEEE.

[18] Mahesh Nanjundappa, Hiren D. Patel, Bijoy A. Jose, and Sandeep K. Shukla. 2010. SCGPSim: A fast SystemC simulator on GPUs. In *2010 15th Asia and South Pacific Design Automation Conference (ASP-DAC)*.

[19] Christian Sauer, Hans-Martin Bluethgen, and Hans-Peter Loeb. 2014. Distributed, loosely-synchronized systemC/TLM simulations of many-processor platforms. In *Proceedings of the 2014 Forum on Specification and Design Languages (FDL)*.

[20] Christoph Schumacher, Rainer Leupers, Dietmar Petras, and Andreas Hoffmann. 2010. parSC: Synchronous parallel SystemC simulation on multi-core host architectures. In *2010 CODES+ISSS*.

[21] Christoph Schumacher, Jan Henrik Weinstock, Rainer Leupers, Gerd Ascheid, Laura Tosoratto, Alessandro Lonardo, Dietmar Petras, and Thorsten Grötker. 2013. legaSCi: Legacy SystemC Model Integration into Parallel Systemc Simulators. In *2013 IEEE International Symposium on Parallel & Distributed Processing*.

[22] Nicolas Ventroux, Julien Peeters, Tanguy Sassolas, and James C. Hoe. 2014. Highly-parallel special-purpose multicore architecture for SystemC/TLM simulations. In *2014 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV)*.

[23] Nicolas Ventroux and Tanguy Sassolas. 2016. A new parallel SystemC kernel leveraging manycore architectures. In *2016 DATE*. IEEE.

[24] Sara Vinco, Valeria Bertacco, Debapriya Chatterjee, and Franco Fummi. 2012. SAGA: SystemC acceleration on GPU architectures. In *DAC*.

[25] Reinhold P Weicker. 1988. Dhrystone benchmark: rationale for version 2 and measurement rules. *AcM SIGPLAn notices* 23, 8 (1988).

[26] Jan Henrik Weinstock. 2023. OpenRISC 1000 Multicore Virtual Platform (or1kmvp). https://github.com/janweinstock/or1kmvp

[27] Jan Henrik Weinstock. 2023. Virtual Components Modeling Library (vcml). https://github.com/machineware-gmbh/vcml

[28] Jan Henrik Weinstock, Rainer Leupers, Gerd Ascheid, Dietmar Petras, and Andreas Hoffmann. 2016. SystemC-link: Parallel SystemC simulation using time-decoupled segments. In *2016 DATE*.

[29] Jan Henrik Weinstock, Luis Gabriel Murillo, Rainer Leupers, and Gerd Ascheid. 2016. Parallel SystemC Simulation for ESL Design. *ACM Trans. Embed. Comput. Syst.* 16, 1 (Oct. 2016). https://doi.org/10.1145/2987374

[30] Jan Henrik Weinstock, Christoph Schumacher, Rainer Leupers, Gerd Ascheid, and Laura Tosoratto. 2014. Time-decoupled parallel SystemC simulation. In *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*.

[31] Hao Ziyu, Qian Lei, Li Hongliang, Xie Xianghui, and Zhang Kun. 2009. A Parallel SystemC Environment: ArchSC. In *2009 15th International Conference on Parallel and Distributed Systems*. IEEE, Shenzhen, China.

[32] Niko Zurstraßen, Ruben Brandhofer, José Cubero-Cascante, Nils Bosbach, Lukas Jünger, and Rainer Leupers. 2024. The Optimal Quantum of Temporal Decoupling. In *2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC)*.