Distributed SystemC Simulation on Manycore Servers

Janne Virtanen, Panu Sjövall, Marko Viitanen, Timo D. Hämäläinen, and Jarno Vanne
Department of Pervasive Computing
Tampere University of Technology
Tampere, Finland
{janne.m.virtanen, panu.sjovall, marko.viitanen, timo.d.hamalainen, jarno.vanne}@tut.fi

***Abstract*—SystemC (SC) is widely used in SoC simulations at various levels of abstraction. The free OSCI SC simulator can only execute on a single core in a sequential manner, which limits the simulation speed. Most speed-up techniques use threading, but this increases synchronization complexity and requires modifying the SC simulator kernel. We propose to use POSIX processes, and call it Inter Process Transaction Level Model (IPTLM) simulation. Our test case is a complete Kvazaar HEVC intra encoder. IPTLM offers 23x speed-up in a 28-core server compared with the standard monocore SC simulation time. IPTLM required manually modifying about 200 SC model code lines compared with the standard SC, which is reasonable when taking the achieved simulation speedup into account.**

Keywords—SystemC, TLM, POSIX, Distributed simulation, Manycore HEVC, Kvazaar

I. Introduction

SystemC (*SC*) [1] has become the mainstream modeling language from transaction level model (TLM) to register transfer level (RTL) abstractions with varying timing accuracy. The basic concepts are SC modules for structural description and SC processes for the behavioral part of the system.

Open SystemC Initiative (OSCI) offers a free SC simulator, which includes a C++ library and can be used with just a compiler and code editor. However, the SC simulator runs only on one *Operating System* (*OS*) process, and the SC scheduler repeats the evaluate-update-notify cycle for each SC process one at a time. Therefore, the simulation execution is inherently sequential and cannot benefit from multiple cores on the simulator computer.

Our goal is to speed-up the simulation without modifying the SC simulator kernel. Our proposal is called *Inter Process Transaction Level Model* (*IPTLM*) simulation, and it is implemented as a new library to SC. IPTLM implements untimed/loosely timed TLM simulation, which is used, e.g., in the design space exploration phase of the SoC design.

In the rest of the paper, we refer to the simulated system as "SC model". "SC process" means SC threads and SC methods in general. By "core" we mean the physical processor core, and by "kernel" the SC simulator kernel.

The main contributions in this paper are:

- A novel parallelization approach for SystemC simulations at OS process level
- New API and functions for SC models
- Speed-up measurements with up to 28 physical cores using a SC model of Kvazaar 4K High Efficiency Video Coding (HEVC) video encoder [2] as a test case

This paper is organized as follows. Section II describes the related work. Section III details our parallel SC implementation and Section IV the SC model for Kvazaar. Section V describes the case studies and results. Section VI concludes the paper.

II. Related Work

Figure 1 depicts an exemplar SC model. Every SC process is executed on an OS user thread, and the whole simulation including the SC kernel itself in an OS process having one OS kernel thread. Only one SC process (OS thread) is run at a time, and it must run to completion before yielding. This ensures deterministic, thread-safe execution, but makes the simulation locked to only one host core at a time.

The two main speed-up techniques either modify the SC kernel to let several SC processes run in parallel, or let several kernel instances run in parallel. We start with the former in the following.

A parallel SC scheduler and mechanisms to synchronize the evaluate-update phase is presented in [6]. They also present four approaches to distribute the SC processes to cores. The best results were achieved with a new SC API, in which the user manually grouped SC processes to cores. The reported simulation time speedup was ~8 for 16 cores.

ParSC [7] has a master-slave OS threads model. The user must modify the SC model to protect against data races for the SC processes executed in parallel in a delta-cycle. The speedup was at best 4.4x in a quad-core cycle-accurate simulation.

legaSCi [12] attempts to avoid any modifications of legacy SC models. It uses Loosely-Timed TLM abstraction and grouping of SC processes to zones sharing the same context. Determinism is achieved by an additional scheduling algorithm for runnable SC processes. However, the SC model still needs to be modified by, e.g., adding some new blocks for inter-zone communication. A speedup of 2.13x is reported on four cores.

Running many kernel instances in parallel is presented in [8]. The authors present a new SC modeling profile called

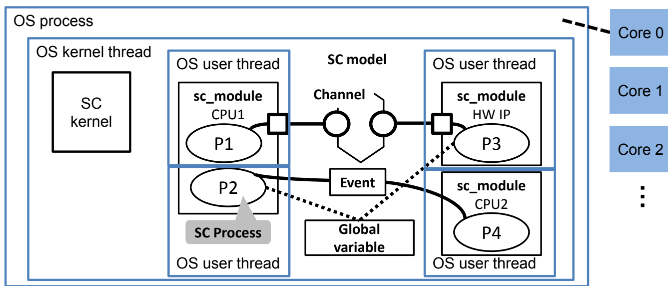

Figure 1. Example SystemC model.

TLM Distributed Time (TLM-DT) and a simulator supporting it based on Portable Operating System Interface (POSIX) Threads. The idea is to remove the centralized simulated time keeping from the kernel. Instead, all SC processes keep it as local, and mutually synchronize time with special messages. A speedup of 1.9x is reported with two cores.

A hybrid solution is presented in [10], in which a two-level hierarchy of threading is utilized. The first layer maps SC processes to different SC kernels, which in turn are mapped to OS kernel-level threads and those to cores. The speedup is 3.3x for four cores, but does not increase anymore by adding more cores.

CoMix [11] is a commercial solution aimed at multi-computer simulation. The SC model is manually cut to modules, which are connected via Transmission Control Protocol / Internet Protocol (TCP/IP) for distributed and loosely synchronized simulation. A speedup of 3.8x is reported in a quad-core virtual machine.

Most of the proposals are based on OS threads, since inter process communication causes larger overhead in general. All proposals consider time synchronization, most delta-cycle ordering of execution to ensure determinism and also load balancing either by static or dynamic SC process allocation.

Unfortunately, it seems that modifications to the SC model cannot be fully avoided, so we accept the fact and only try to limit the developer effort. Another issue is that parallel SC simulators are not standardized as is the OSCI SC simulator, which complicates long-term tools maintenance.

Our constraints for parallel SC simulation are: i) as standard toolset as possible, ii) moderate manual work to modify the SC models, and iii) scalability to dozens of cores. We are not attempting a perfect core load balancing if the total simulator time is sufficiently decreased as we add more cores.

III. IPTLM

Distributing the SC simulation to OS processes means technically many independent SC simulators, which has two requirements. First, the distribution of the original SC model to the kernels. Second, the execution coordination between the kernels at runtime. Ideally, both of them are automated and the user does not need to modify the original SC models of the simulated system.

Our IPTLM is based on manual distribution like most of the related work. In addition, we only consider untimed/loosely timed TLM, and primarily aim at reducing the simulator wall clock time.

Our approach sets the following limitations to the SC model. The main requirement is a strict separation of computation and communication that takes place only using channels between the SC modules. SC events are not allowed for signaling between the SC modules. However, events may be used if all related SC processes are executed within one kernel, but this requires manual work every time the distribution is changed. Therefore, we completely avoid the use of events. The channel communication abstraction is TLM 2.0.

We recognize two levels of parallelization. The first is the way the SC model describes parallel execution of the application. This concerns the mapping of SC processes to the SC modules, and we may have dozens of SC models for exploring the alternatives. The second is distribution of the SC modules to kernels. The number of kernels per core is configured separately. As a whole, we may change the mapping of SC processes to SC modules, the SC modules to kernels, and kernels to cores.

The SC model is manually modified for IPTLM as follows. The standard SC TLM channel related functions are replaced by the IPTLM wrapper functions whenever the other SC process is executed in another kernel. Alternatively, the underlying implementation of the standard TLM functions would be automatically substituted, but this is out of scope of this paper.

A. IPTLM architecture

IPTLM is based on POSIX processes and POSIX shared memory for inter process communication. POSIX queues and messages could be used as well, but shared memory is feasible for most standard computers used in simulations.

Figure 2 depicts an overview of IPTLM and the write sequence. IPTLM has separate classes and interfaces for a master and slave bus interface in the SoC SC model. One slave is connected to one master as point to point. They connect when their objects are created in SC. They will identify each other with a constant string defined at SC model creation time. For the underlying technology (POSIX shared memory) the name of the memory file is the same constant string. IPTLM utilizes semaphores to synchronize between the master and slave processes.

The bus master initiates either read or write (1. - 4.) and the slave waits for a request (6. - 7.). Both are blocking calls. When a request is received, the slave has a chance to respond based on whether or not it was a write or read and which address was targeted by the master (8. - 9.). The slave then makes a transfer, and releases the master (10. - 12.), thus releasing both (13. - 16.). The sequence for reading is similar, except the master executes *memcpy()* after slave has transferred (11.-13.). Function *transfer()* works for both transfer directions.

IPTLM provides a C++ library extension to the standard SC API to mediate between the POSIX processes. In the current implementation, this is lightweight and consists of six functions as summarized in Table 1.

Like TLM, IPTLM is completely layered on top of the SC kernel. IPTLM API does not directly correspond to TLM API. The main difference is that the slave does not register *b_transport* functions, but instead calls the function *slave::wait_request* when it is ready to receive a request from a master. If the master has already sent a request, the function returns immediately. After that, the slave inspects the request, and transfers accordingly.

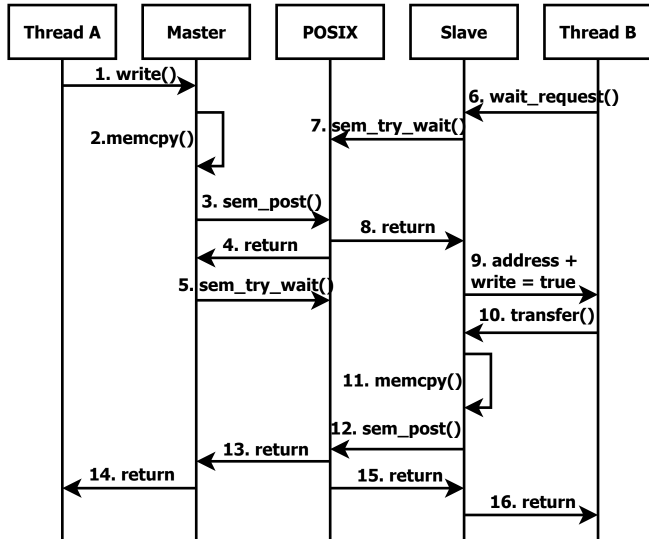

Figure 2. IPTLM write sequence.

Table 1. IPTLM functions.

IPTLM function	Description
master::master	Creates a new master object, establishes link with the corresponding slave.
master::write	Writes contents of the buffer parameter to slave.
master::read	Writes data coming from slave to the buffer parameter.
slave::slave	Creates a new slave object, establishes link with the corresponding master.
slave::wait_request	Waits for a request from the corresponding master, returns true, if master is writing, else false.
slave::transfer	Executes the transfer initiated by the master.

B. IPTLM performance

We measured the IPTLM data transfer performance with a minimal system model that includes only one master and one slave and very simple application just sending data in a one-way manner with protection (semaphore). To compare, we have three models: standard SC TLM model using *b_transport*, IPTLM, and pure POSIX shared memory. We used a PC with i7-4790X@3.60 GHz with CentOS 6.6 guest hosted by Windows 7 in this test.

Naturally the TLM model is the fastest, 3.4 GiBps using 1kiB transfers, since there is no operating system process overhead and data copying is by reference. IPTLM and POSIX achieve 40 MiBps and 50 MiBps. With larger transfer size, 100 kiB, IPTLM achieves 4.3 GiBps. Thus, IPTLM favors either computation intensive or large transfer sized simulations. In the following, we present the results in a real test case. For brevity, we consider only the simulator time and omit details of simulated computation and communication times.

IV. Test Case

As a real-scale test case, we use an open-source Kvazaar HEVC intra encoder, version 0.4.2 [2]. The C source code of Kvazaar is modified to SystemC model for hardware architecture exploration. Comparing potential HW architectures for Kvazaar is out of the scope of this paper, but we focus on speeding up the simulation of any SC model of Kvazaar intra encoder. Here, we have chosen one test case, where Kvazaar is run under *All-Intra* (*AI*) coding configuration with the following command line options:

```
--input-res 3840x2160 --no-rdoq --no-sao --no-deblock -q 18 --rd 2 -p 1 --full-intra-search -n Frames --wpp --owf N,
```

where *rate distortion optimized quantization* (*RDOQ*), *sample adaptive offset* (*SAO*), and a deblocking filter are disabled. The tested 4K video sequence is “Bosphorus” [3] (600 frames) with *quantization parameter* (*QP*) value of 18. RDO level (*rd*) is set to full, intra period (*p*) is every frame, and an exhaustive intra search is enabled. The intra coding tools of Kvazaar are detailed, e.g., in [4].

Encoding parallelism can be exploited by running several *Coding Tree Units* (*CTUs or LCUs*) of the same picture in parallel. This can be done through a *wavefront parallel processing* (*WPP*) which exploits CTUs in already encoded regions. *Overlapped Wavefront* (*OWF*) processing brings the exploitation to the highest end by dealing with several frames in parallel. In our experiments, one, three, or eleven frames are coded in parallel, i.e., $N = \{0, 2, 10\}$. Visualization of OWF in Kvazaar can be seen in [5].

A. Standard SC model

This acts as the purely sequential reference model to which we compare IPTLM. Figure 3 depicts a simplified block diagram of the Kvazaar SC model used in this paper. The major parts of Kvazaar execution are divided into subsystems *search_cu* and the rest is kept in a SC module *kvazaar_core* acting as a master process. All communication between the SC modules takes place using blocking TLM *b_transport*, and no events are used between SC modules.

The master module delivers data to slave modules, collects the results and performs some common tasks like bit stream assembly. The intra prediction subsystems are acting as slave modules *acc_x*, including SC processes *search_cu*, one or more for each wavefront and frame.

The number of instantiated SC modules depends on Kvazaar coding settings (OWF) and video resolution. 4K video frames are large enough to instantiate even tens of slaves. As a whole, we can explore different parallelization setups by this master-slave model.

B. *IPTLM SC model*

The above standard SC model acts as a reference for parallelization of the SC simulation. The parallelized IPTLM model is obtained by replacing the *b_transport* functions by the IPTLM functions.

Listing 1 illustrates the usage of IPTLM in SC module's master interface side. For comparison, there are also the TLM code excerpts to write/read at the specified address of the slave, using the specified buffer of the master. A convenience function was used to reduce redundant code. The corresponding IPTLM transfers also receive buffers as parameters, although in this case addresses are omitted, since unlike in TLM, the slave side receives the requests sequentially with other functionality. The argument *i* is related to time stamping instrumentation for simulator time reports.

V. MEASUREMENTS

Our host computer includes two 14-core Intel Xeon E5-2697v3 2.6 GHz CPUs with 32 GB RAM. The OS is Ubuntu natively executed without any virtual machine.

The simulator time is acquired by instrumentation code available in the original Kvazaar source code and by the IPTLM functions. We log timestamps for computation, communication (POSIX *memcopy* and *semaphores*) and idle waiting for all SC processes. The CPU load is measured using *mpstat* [13].

The correctness of the encoded video in parallel simulation was verified by comparing the encoder output to a non-parallelized native Kvazaar encoder using the same video sequence and parameters. The PSNR is recorded for each

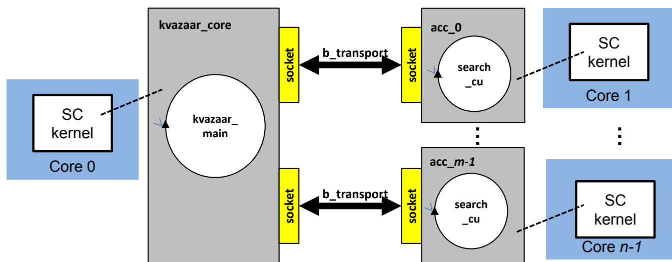

Figure 3. High-level view of Kvazaar SystemC model and execution mapping on host cores.

Listing 1. Example replacement of b_transport by IPTLM.

```
//Original done with b-transport
    transport( MISC_DATA_ADDR, (unsigned char*)buffer, BUF_SIZE,
    true );
    transport( ENCODER_STATE_ADDR, (unsigned
    char*)state->encoder_control, sizeof(encoder_control_t), true
);
    transport( LCU_ADDR, (unsigned char*)lcu, sizeof(lcu_t), true
);
    transport(  LCU_ADDR,  (unsigned  char*)lcu,  sizeof(lcu_t),
false );
//Convenience function
    void transport( int address, unsigned char* buffer, size_t
len,
    bool write ){
    ...
//Blocking transport call
    kvazaar_global->socket->b_transport( *trans, delay );}
//The same done with IPTML
    master->write( i, buffer, BUZ_SIZE );
    master->write( i, (void*)state->encode_control,
    sizeof(encoder_control_t) );
    master->write( i, (void*)lcu, sizeof(lcu_t) );
    master->read( i, (void*)lcu, sizeof(lcu_t) );
```

experiment.

A. *Setting up parallel SC simulation*

The standard and IPTLM SC models already define the mapping of SC processes to SC modules (application level parallelism), so the task is to map the SC modules to the kernels and further the kernels to the cores for parallel simulation. Exploration of different SC processes and module mappings are out of the scope of this paper.

Figure 3 depicts the mapping used in this paper. We assign one or more SC processes to one SC module, one SC module to one kernel, and one kernel per core. This is carried out in a code editor or graphically drag-dropping the SC modules on top of kernel placeholders in *Kactus2* tool [9]. Independent of the input method, the mapping information is written to a header file copied to all kernels.

The master process includes one communication thread for each slave. The slave specific thread is always executed on the same core as the slave process, but otherwise the master is executed in its own core. Thus, the number of slave processes is one less than the number of available cores, e.g. 27 slaves in a 28-cores machine. A shell script is used to launch each simulation experiment. Automatic load balancing is not used.

B. *Results*

Figure 4 depicts the total simulator wall-clock time and average wait times for the master and slave processes as a function of the cores for OWF={0,2,10}. The first column, labelled as "ref", plots the simulator time for the reference SC model and the remaining ones for the IPTLM with 1-28 cores. The reference model simulator time is 20.037s (OWF=0), 20.089s (OWF=2) and 20.132s (OWF=10), which is slightly lower than that of the IPTLM with one core. This is explained by the OS process overhead in IPTLM, but IPTLM starts to quickly pay back after three cores. It should be noted that the OS and SC kernel have freedom to execute the reference SC model on multiple cores if possible, but in practice only one core is used.

Without overlapping frame encoding (OWF=0), the execution time flattens after 20 cores. In this case, the 4K frame does not contain enough data to fully exploit all the cores since we do not have automatic load balancing. With three (OWF=2) and eleven (OWF=10) frames encoded in parallel, the simulator time continuously drops towards the 28 cores.

The identical results for OWF2 and OWF10 mean that the cores are already fully occupied when three frames are encoded simultaneously. The PSNR in all the experiments were attempted to be same, which resulted in on average 45.9 (Y) 49.2 (U) 48.6 (V). Figure 5 depicts the simulator time speedups. The graphs confirm almost linear scalability for OWF=2 and OWF=10, and the saturation effect after 20 cores for OWF=0.

To analyze the results in more detail, Figure 4 plots the master and slave process idle times as well. Wait times have been measured from the master process communication threads and slave processes individually, and averaged by the number of slaves.

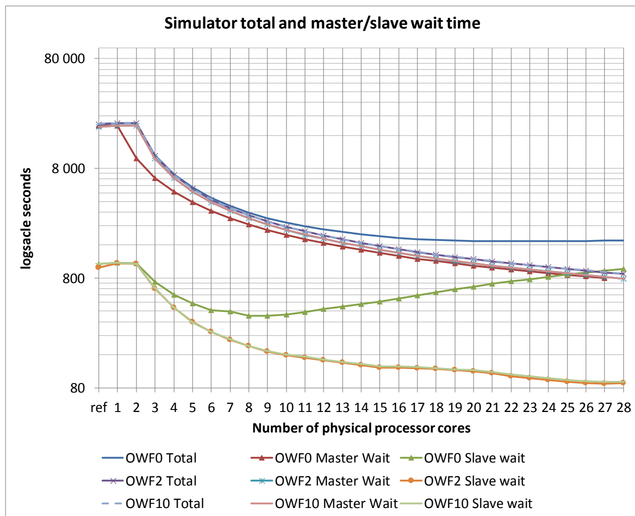

Figure 4. Simulator wall-clock time and average wait times for the master and slave processes.

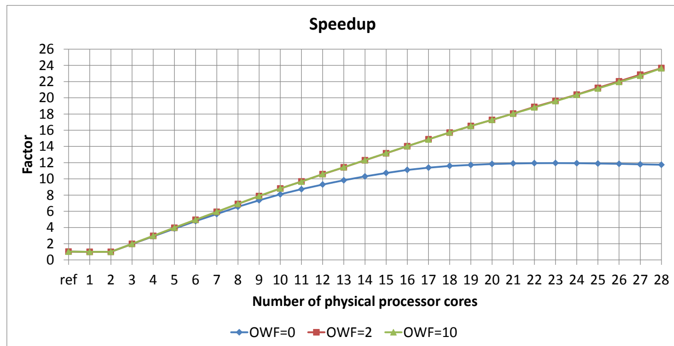

Figure 5. Simulation speedup.

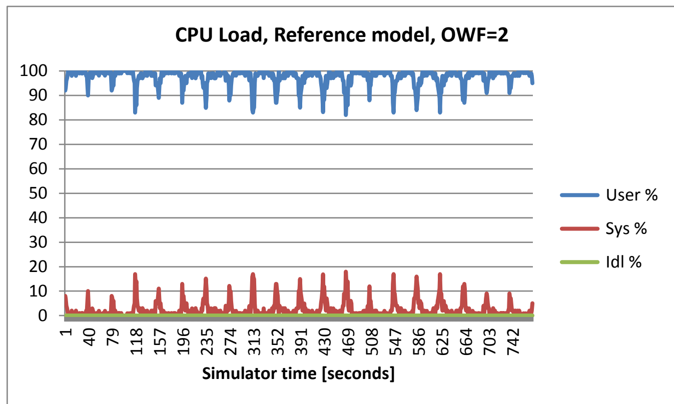

Figure 6. CPU load of the reference model.

With OWF=0, the master roughly waits for 90% of the time, and slaves on average for around 10% of their total time up to the saturation point. For 28 cores, the waiting percentages are 46% and 55%, respectively, showing shortage of data to all cores. For OWF=2, the master and slave waiting percentages are 91% and 10% for 28 cores, and 93% and 5% for one core. The master is clearly not choking the execution in any

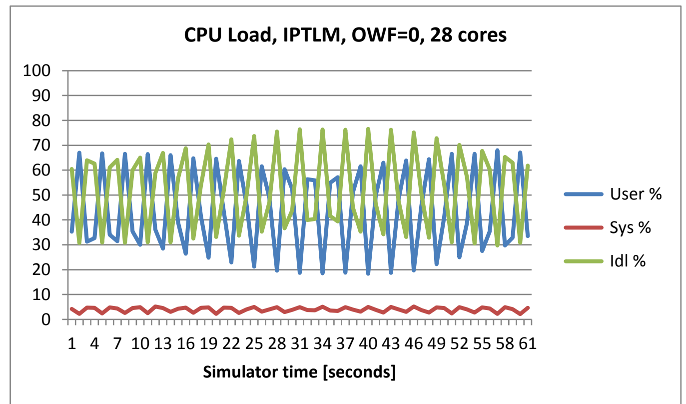

Figure 7. Load when encoding one frame at a time.

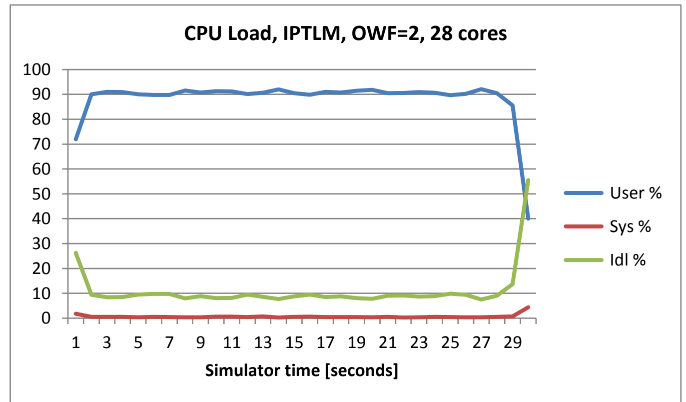

Figure 8. Load when encoding three frames at a time.

configuration, which was the design goal for the original parallel Kvazaar SC model.

Figure 6 depicts the CPU load for the reference model with OWF=2 for 20 frames in Bosphorus. For OWF=0, it was almost constantly 100%. The attempt to simulate encoding of three frames in parallel causes significant system call overhead and slows down the simulation. Clearly this is not feasible for standard SC simulation. It should be noted that the load is measured for the active cores, in this case one core, while others are idle and not counted in to the load figures.

Figure 7 depicts the CPU load for IPTLM with OWF=0. The shape of the idle waiting clearly shows the starving for data for 28 cores.

Figure 8 plots the load with OWF=2. The ratio between the user and system level load is very good, which shows that IPTLM is not causing exhaustive number of system calls, which is the danger in OS process based simulation distribution.

C. Effort

Parallelizing the Kvazaar source code for simulation on multiple cores took about one week and involved replacing the existing TLM-communication with IPTLM, which finally affected about 200 lines of code. The most laborious task was to rewrite the code for events. SC_events were replaced with

one bit signals, because IPTLM does not support events between modules.

D. Comparison

To the best of our knowledge, there is no related work combining both parallel SystemC simulation and parallel HEVC encoding. Thus, we compare the results to both of them separately.

Our speedup scales very well compared to the related parallel SystemC simulations, which are in the order of 1.8×-4× compared to sequential simulation. This is because the overhead (system and idle times) is kept moderate compared with even thread-based proposals that are generally more lightweight than process-based parallel execution.

The speedup is very good also compared with the related work on parallel HEVC encoding, which is listed in Table 2. The last column lists the ratio at which the cores could have been utilized, 100% being the ideal. As a whole, we succeeded also in the Kvazaar parallelization approach, even though the primary scope of this paper was how to speed up SystemC simulations. For this reason, a more comprehensive encoder comparison is excluded in this paper.

Table 2. Comparison of max speedup in parallel HEVC.

Ref	Resolution	QP	CPU	Core	Speedup	Ratio
[14]	2560x1600	27	E5-2670	8	5,5	69 %
[15]	1920x1080	37	GX36	36	17,0	47 %
[16]	3840x2160	30*	E5-2699	36	31,9	89 %
[17]	1920x1080	27	Opt.6272	36	21,9	61 %
This	3840x2160	18	E5-2697v3	28	23,6	84 %

*average

VI. Conclusions

This paper presented a new approach called Inter Process Transaction Level Model (IPTLM). It uses POSIX processes to support parallel simulations using standard sequential SystemC kernel. Even though a direct comparison to the related work is difficult due to different test cases and unavailability of open thread-based parallel SystemC simulators, our speed-up results show very good scalability over state of the art.

The associated coding effort was about one week with 200 lines of new or modified code, which is moderate compared to the time for running simulations. The experiments in this paper took 173 hours in wall clock time on our computer, and still we used only one possible parallel Kvazaar architecture with one test sequence and three different encoder configurations. To perform thorough design space explorations, each of these dimensions can be multiplied, which means that the obtained speed up will have significant time savings in the future design space explorations.

References

[1] IEEE Standard for Standard SystemC Language Reference Manual, *in IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)*, Jan. 2012.

[2] *Kvazaar HEVC encoder* [Online]. Available: https://github.com/ultravideo/kvazaar

[3] *Ultra video group* [Online]. Available: http://ultravideo.cs.tut.fi/

[4] M. Viitanen, A. Koivula, A. Lemmetti, J. Vanne, and T. D. Hämäläinen, "Kvazaar HEVC encoder for efficient intra coding," *in Proc. IEEE*, Lisbon, Portugal, May 2015, pp. 1662-1665. *International Symposium on Circuits and Systems*

[5] *Kvazaar Visualizer* [Online]. Available: http://ultravideo.cs.tut.fi/#visualizer

[6] P. Ezudheen, et al., "Parallelizing SystemC Kernel for Fast Hardware Simulation on SMP Machines," *in proc. ACM/IEEE/SCS Workshop on Principles of Advanced and Distributed Simulation*, pp. 80-87, 2009.

[7] C. Schumacher, et al., "parSC: synchronous parallel systemc simulation on multi-core host architectures," *in Proc. IEEE/ACM/IFIP The International Conference* on *Hardware/Software Codesign* and *System Synthesis*, pp. 241-246, 2010.

[8] A. Mello, et al., "Parallel simulation of systemC TLM 2.0 compliant, MPSoC on SMP workstations," *in Proc. Design, Automation & Test in Europe Conference & Exhibition*, pp 606-609, 2010.

[9] Kactus2 [Online]. Available: http://funbase.cs.tut.fi

[10] M. K. Chung, J. K. Kim, and S. Ryu, "SimParallel: A high performance parallel SystemC simulator using hierarchical multi-threading," *in Proc. IEEE International Symposium on Circuits and Systems*, Melbourne, Australia, Jun. 2014, pp. 1472-1475.

[11] C. Sauer, H-M. Bluethgen, and H-P. Loeb, "Distributed, loosely-synchronized SystemC/TLM simulations of many-processor platforms", Specification and Design Languages, Vol. 978, 2014.

[12] C. Schumacher, J. H. Weinstock, R. Leupers, G. Ascheid, L. Tosoratto, A. Lonardo, D. Petras, and A. Hoffmann., "legaSCi: Legacy SystemC Model Integration into Parallel Simulators," *ACM Transactions on Embedded Computing Systems* vol. 13, no. 5, Nov. 2014, pp. 165:1-165:24.

[13] Command mpstat, Linux User's Manual [Online]. Available: http://www.linuxcommand.org/man_pages/mpstat1.html

[14] Yanan Zhao, Li Song, Xiangwen Wang, Min Chen & Jia Wang, "Efficient realization of parallel HEVC intra encoding", *International Conference on Multimedia and Expo 2013*

[15] S. Zhang, X. Zhang & Z. Gao, "Implementation and improvement of Wavefront Parallel Processing for HEVC encoding on many-core platform", *International Conference on Multimedia and Expo* 2014

[16] Z. Wen, B. Quo, J. Liu, J. Li, Y. Lu & J. Wen, "Novel 3D-WPP algorithms for parallel HEVC encoding", *International Conference on Acoustics, Speech and Signal Processing* 2016, pp. 1471.

[17] K. Chen, J. Sun, Y. Duan & Z. Guo 2016, "A Novel Wavefront-Based High Parallel Solution for HEVC Encoding", *IEEE Transactions on Circuits and Systems for Video Technology,* vol. 26, no. 1, pp. 181-194