Port Call Path Sensitive Conflict Analysis for Instance-Aware Parallel SystemC Simulation

Tim Schmidt, Zhongqi Cheng, and Rainer Dömer
Center for Embedded Computer Systems
University of California, Irvine, USA

Abstract— Many parallel SystemC approaches expect a thread safe and conflict free model from the designer. Alternatively, an advanced compiler can identify and avoid possible parallel access conflicts. While manual conflict resolution can theoretically be more precise, it is impractical for real world applications because of the inherent complexities. Here automatic compiler-based analysis is preferred which provides conservative conflict avoidance with minimal false positives. This paper introduces a novel compiler technique called *Port Call Path* analysis that greatly reduces the amount of false positive conflicts resulting in significantly increased simulation speed. Experimental results show that the new analysis reduces the amount of false conflicts by up to 98% and, on a 4-core processor, speeds up the simulation up to 3x for a NoC particle simulator and 3.5x for a bitcoin miner SystemC model.

I. Introduction

Embedded systems are part of our daily life, visible as a cell phone or invisible in a combustion engine in a car. The high demand of constantly increasing functionality of these products is associated with more complex design processes. Designers use simulations as a tool to make better design decisions for their prototypes. However, simulations of complex systems are time-consuming and become a bottleneck in the tool flow.

SystemC [1] has been established as the de-facto and official Accellera standard for modeling and simulating of embedded systems. The official IEEE proof-of-concept simulator runs simulations in a sequential fashion. This means only one simulation thread is active at any time during the simulation, also if many design parts could be simulated in parallel. Consequently, the simulator can use at most one core on multi and many-core host simulation platforms.

SystemC TLM 2.0 is a library-based approach to speed up the simulation where simulation threads are temporally decoupled. Specifically, the designer defines manually a time quantum in which a thread performs without performing any synchronization points. Unfortunately, the gained speedup comes with the disadvantage of simulation inaccuracy [2]. Other works such as [3] and [4] provide a parallel simulation kernel where the user has to manually translate the sequential design into a parallel design. In detail, the individual simulations threads must be analyzed for conflicting variable access. An overlooked conflict can compromise the simulation and let the simulation fail.

A. Problem Definition

A parallel SystemC simulator needs the information which segments of a simulation thread can be executed in parallel. One option is to provide this information in form of a conflict table. A table entry must be *true* if two segments have a potentially conflicting variable access, otherwise the entry should be *false*, allowing parallel execution.

More specifically, a table entry is classified into four categories: true positive (TP), true negative (TN), false positive (FP), and false negative (FN). The first two categories TP and TN describe a correctly marked positive variable access conflict, respectively, no access conflict. A FP is a table entry where a conflict is marked, however, no actual conflict exists. A FN is a table entry where no conflict is marked, however, a conflict exists. This kind of a table entry is not allowed because it compromises the simulation.

The conflict table generation can be done in two ways, manually or automatically. On one hand, the manual analysis can eliminate many FP and FN entries because the designer can utilize application knowledge. However, this analysis is very time consuming and is not applicable for real world examples. On the other hand, the compiler driven automatic analysis can be done in a few minutes. However, a compiler cannot be as precise as the manual analysis, e.g. due to static pointer analysis. Consequently, a compiler must be *conservative* to safeguard the simulation correctness. In other words, table entries are marked as a potential conflict where actually no conflict exists. This behavior causes FP entries and limits the parallel simulation performance.

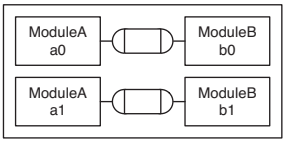


Fig. 1: Two pairs of communicating modules

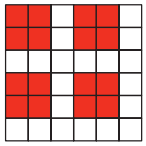


Fig. 2: Data conflict table

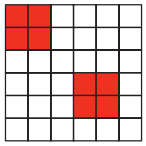


Fig. 3: PCP-sensitive data conflict table

Fig. 1 shows an example where two pairs of a sender and a receiver exchange data. The corresponding tables of conflicting variables accesses (red filled) are shown in Fig. 2 and Fig. 3. The left table is created with a conservative conflict analysis. Often, the parallel simulator can run only one simulation thread because of FP conflicts. The right table

978-3-9819263-0-9/DATE18/©2018 EDAA

is built with a more precise conflict analysis. In this simple example 50% of the positive marked conflicts are FP conflicts. Consequently, the simulator can run more threads in parallel. Our RISC compiler [5] instruments the conflict table in design file. Later, the parallel SystemC simulator uses the conflict table for scheduling decisions.

In this paper, we extend the static analysis capability of the SystemC compiler [5] to largely reduce the FP entries in the conflict table. Specifically, we introduce the *Port Call Path* (PCP) technique to have more precise context information of accessed variables. During the needed segment graph analysis, we take the related port call history into account. This information allows us to distinguish between individual variables in channels instead of clustering them.

B. Related Work

Parallel discrete-event simulation (PDES) [6] is a well-studied subject. The segment graph data structure for PDES was first introduced for synchronous and out-of-order fashion in [7]. This concept is extended with instance IDs for modules to have higher precision in the conflict analysis in [8]. An extension for thread and data level parallelism is in [9]. In contrast to these works, our new Port Call Path technique for segment graphs allows a more precise analysis for variable accesses in channels to reduce FP conflicts. Also, in contrast to [10], our analysis is purely static.

The concept of time decoupling is implemented in the SystemC TLM2.0 library where threads execute for a user defined quantum in an unsynchronized manner. The missing synchronization between threads limits the number of context switches to gain higher simulation speed. However, the simulation boost is associated with the price of lower accuracy [2]. The works in [4] and [11] propose techniques to parallelize time decoupled designs for multi-core systems. The designer must manually partition the design into a thread safe model which is conflict free. So an overlooked conflict can compromise the simulation and leads to simulation failures. The authors in [3] describe an API to transform manually a sequential SystemC design into a parallel design. Compared to these works, our compiler driven approach automatically identifies race conditions and instruments the design. In other words, the transformation does not require application-specific knowledge or manual modeling.

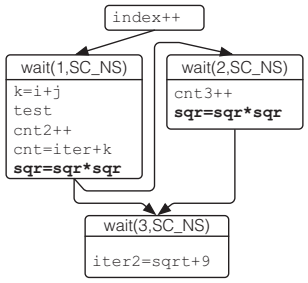
In [12] static analysis of SystemC models is done to use GPUs as a simulation platform. In contrast, our static analysis uses module and channel instance IDs to distinguish channel variables. Other works in context of parallel SystemC are in [13]. They are using a modified version of the of the Chandy-Misra-Bryant algorithm for their analysis. In comparison, we provide a new analysis to have precise analysis for object instances.

II. Conflict Analysis

A. Segment Graph

The segment graph is a data structure to partition the individual simulation threads of a design in smaller pieces. Later, we use this graph to identify potential race conditions and notification dependencies in the design. In detail, for each simulation thread we partition the source code into segments and link them respectively to the control flow. A segment includes all statements between two scheduling steps. The transition from the application domain back to the scheduler domain happens through a `wait()` function call. Fig. 4a shows some example source code and the related segment graph in Fig. 4b.

```
0 void foo() {
1   index++;
2   wait(1,SC_NS);
3   k=i+j;
4   if(test){
5     cnt2++;
6     wait(2,SC_NS);
7     cnt3++;
8   } else {
9     cnt=iter+k;
10  }
11  sqr=sqr*sqr;
12  wait(3,SC_NS)
13  iter2=sqr+9; }
```



(a) Source Code (b) Segment Graph

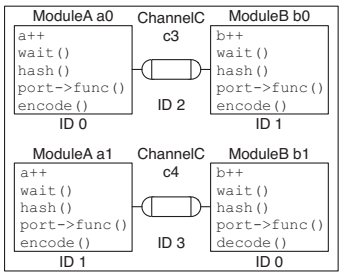
Fig. 4: Example of a Segment Graph [9]

Fig. 4b shows the difference to the traditional control flow graph and a segment graph. The segment graph has four segments, one initial and three related to `wait()` function calls. The statement in Line 11 `sqr=sqr*sqr` appears in the two segments from Line 2 and Line 6. This is possible because both can reach Line 11. Note that a segment graph is generated per module definition and not per module instance.

B. Variable Access Analysis

The data conflict analysis compares all pairs of segments for potential race conditions. Basically, two steps are needed to identify a conflict between two segments. First, we create two sets of read and written variables for each segment. In particular, we traverse each expression of a segment and identify the accessed variable symbols. Second, we have to check if there is a read-write or write-write dependency between the two segments. For instance, in Fig. 5a, the first segment of ModuleA and ModuleB contains only the symbols of `a` and `b`, respectively. Consequently, the segments do not cause a race condition and can be simulated in parallel.

The situation is different for the module instances `a0` and `a1`. The segment graph is generated per thread and not per instance. Consequently, both instances are represented by the same segment graph. So, the symbols `ModuleA::a` and `ModuleA::a` are in conflict because the segment graph stores only the accessed symbol. This problem can be solved through *module instance IDs* [8]. Each module has a unique ID which is defined due to the declaration order and its type in the source file. Fig. 5b shows the declaration and related IDs for the design in Fig. 5a. Now, the conflict analysis considers segments in context of a module instance ID. So we can distinguish between the module instances and their members.

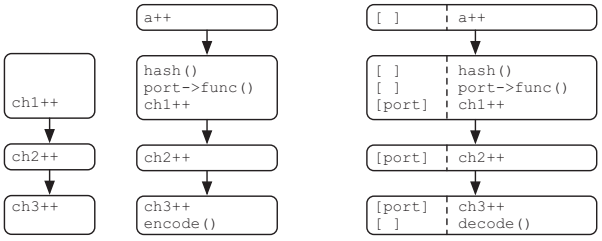


(a) Module details

```
SC_MODULE(Top) {
  ChannelC c1; // ID 0
  ChannelC c2; // ID 1
  ChannelC c3; // ID 2
  ChannelC c4; // ID 3
  ModuleA a0; // ID 0
  ModuleA a1; // ID 1
  ModuleB b1; // ID 0
  ModuleB b0; // ID 1
  ...
};
```

(b) Source code

Fig. 5: Additional module details of Fig. 1



(a) Channel only (b) Without Port Call Path (c) With Port Call Path

Fig. 7: Segment Graph of Fig. 6 for ModuleA

C. Limitations of Module Instance IDs

Modules communicate to other modules via channels to exchange data. Technically, ports are the gateways to channels and they are the linking component between module and channel. So, through a *port call*, the control flow leaves the scope of a module and enters the scope of a channel. Exemplary, Fig. 6 shows the transitioning control flow of a simulation thread from a module into a channel and back.

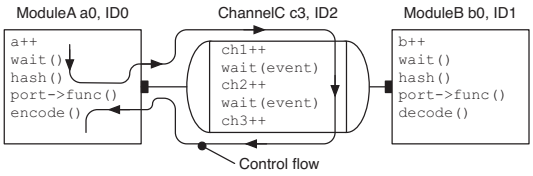


Fig. 6: Additional channel details of Fig. 5

A segment contains symbols of module and channel members at the same time. The union of these symbols occurs only because segment boundaries are defined as entry points to the scheduler. So, a port call does not start a new segment and is interpreted as a regular function call. Consequently, a segment contains the union of expressions which come from modules and channels. This is demonstrated in Fig. 6 where a module communicates through a channel. The associated segment graph in Fig. 7b shows expressions from the module (e.g. `hash()`) and channel (e.g. `ch1++`) in the second segment.

The conflict analysis in context of channels with only module instance IDs is more challenging. In detail, the function HASACCESSCONFLICT$(\text{Seg}_1, \text{Id}_{S1}, \text{Seg}_2, \text{Id}_{S2})$ determines if two segments have a conflict. The instance ID parameters are the module instance IDs where the simulation thread of the segment is spawned. So, all segment members are interpreted with the given module instance ID. However, this interpretation is not valid for channel variables. In Fig. 6, module `a0` and module `b0` are bound to channel `c3`. Segment 2 starts after the `wait()` call in ModuleA and includes the channel expression `ch1++`. Similarly, Segment 4 starts after the `wait()` call in ModuleB and also includes the channel expression `ch1++`. Segment 2 interprets the variable `ch1++` with instance ID 0 and segment 4 interprets the variable `ch1++` with instance ID 1. This gives the impression that both modules are not connected to the same variables. So, the result would be a false negative conflict which is not allowed. Correctly, the variable `ch1++` must be interpreted with instance ID 2 following the declaration order in Fig. 5b.

The segment graph stores only the read and write access of the potentially used symbols of a segment. A naive approach is a scope analysis of each symbol. If a symbol is declared inside a channel, the instance ID will be mapped to a dedicated instance ID for all channels. However, this strategy has two strong limitations for the speed of parallel simulation.

First, all channel variables get the same instance ID. So, all channel variables are in conflict with each other. As a result, the parallel communication between modules happens in a sequential fashion for the entire simulation.

Second, the sequential communication affects the parallel communication of modules. The segment where the port call takes place and the segment where the port call returns inherit the conflicts from the channel. This is illustrated in Fig. 6 and Fig. 7b where ModuleA communicates via ChannelC. The second segment includes the function `hash()` as well as the increment expression of the channel member `ch1++`. The last segment includes the function `encode()` and the increment expression of the channel member `ch3++`. Both segments can only be executed in parallel to other segments when there is no other communication during that simulation cycle.

III. PORT CALL PATH SENSITIVE SEGMENT GRAPHS

Our proposed *Port Call Path* (PCP) analysis for channels allows to distinguish individual channels through *channel instance IDs* without modifying the design. In other words, the two communicating parties in Fig. 5a can then execute in parallel without interfering with each other.

A. Port Call Analysis

For this advanced approach, the conflict analysis needs more context information. So, we store for each expression additionally the PCP history. The segment graph with and without PCP for the example Fig. 6 is shown in Fig. 7b and Fig. 7c. Particularly, the PCP can be a single port call or a list of port calls, e.g. through hierarchical channels communication.

The function HASACCESSCONFLICT$(\text{Seg}_1, \text{Id}_{S1}, \text{Seg}_2, \text{Id}_{S2})$ determines if two segments have a variable access conflict.

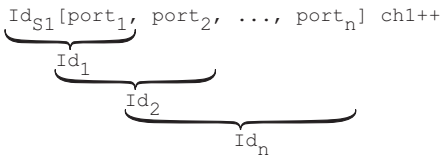


Fig. 8: Translating module instance ID to channel instance ID

Initially, every expression of a segment is interpreted with the related instance ID function parameter. However, the instance ID for channel expressions must be updated for the later conflict analysis. Exemplary, Fig. 8 shows the expression `ch1++` with the associated PCP (port$_1$, ..., port$_n$) of segment Seg_1. A central tool to determine the channel ID is the API of the RISC compiler [5]. Specifically, we have created the function GETCHANNELID(port, id) to determine for a given port and instance ID the instance ID of the mapped channel. The parameter `id` describes the instance ID of the enclosing object for the parameter `port`. For hierarchical mappings, first, Id_1 is computed via GETCHANNELID(port$_1$, Id$_{S1}$). Afterwards, Id_i is computed via GETCHANNELID(port$_i$, Id$_{i-1}$). The function TRANSLATECHANNELID(Id$_{S1}$, GETPCP(s)) includes all steps to compute the final channel ID.

B. Advanced Access Analysis

The algorithm to detect access conflicts between two segments is shown in Algorithm 1. Essentially, we compute for each variable symbol two attributes to store the read and write instance ID context. Let's say a segment has read access to the instances 1, 2, and 3 of variable `x`. Also, there is no write access to any instances of variable `x`. The resulting sets are $x_{\text{READ}} = \{1, 2, 3\}$ and $x_{\text{WRITE}} = \{\}$. Before an instance ID is added to a set, channel context analysis is necessary. If the symbol is in a module, its instance ID applies. If a symbol is declared in a channel, we determine the related channel ID. For instance this is in Line 5 where we use the function TRANSLATECHANNELID().

Overall, the algorithm is divided into two steps. In the first step, the read and write analysis of segment seg_1 takes place. All symbols are marked with their access type.

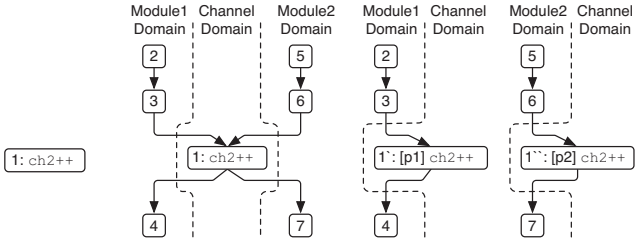
In the second step, a similar analysis happens. However, instead of adding a new read or write instance ID to the symbol, an access check is performed. In Line 21 we check if a read-write conflict exists, i.e. if the read symbol of seg_2 is written by seg_1. In detail, we check if the symbol is already marked with the same instance ID. If this is the case, we return `true` because a conflict exists. Similarly, in Line 30 we check for a write-write conflict. We iterate over the written symbols of segment seg_2. If the symbol is already marked as read or written, a conflict exists.

C. Segment Graph Generation with Port Call Context

The classic segment graph is generated in two steps. In the first step, all channel functions are analyzed. For each function a partial segment graph is built as in Fig. 9a. In the second step, the individual simulation threads are analyzed. When a port call takes place, the pre-analyzed segments of the channel are reused and *linked* as in Fig. 9b. Consequently, channel segments are shared between multiple simulation threads, no individual PCP can be associated, and channel variables cannot be distinguished by module IDs.

Algorithm 1 Conflict analysis between two segments

```
1: function HASACCESSCONFLICT(seg_1, id_1, seg_2, id_2)
2:     for all s ∈ GETREADSYMBOLS(seg_1) do
3:         id ← id_1
4:         if ISCHANNELSYMBOL(s) then
5:             id ← TRANSLATECHANNELID(id_1, GETPCP(s))
6:         end if
7:         S_READ ← S_READ ∪ {id}
8:     end for
9:     for all s ∈ GETWRITESYMBOLS(seg_1) do
10:        id ← id_1
11:        if ISCHANNELSYMBOL(s) then
12:            id ← TRANSLATECHANNELID(id_1, GETPCP(s))
13:        end if
14:        S_WRITE ← S_WRITE ∪ {id}
15:    end for
16:    for all s ∈ GETREADSYMBOLS(seg_2) do
17:        id ← id_2
18:        if ISCHANNELSYMBOL(s) then
19:            id ← TRANSLATECHANNELID(id_2, GETPCP(s))
20:        end if
21:        if id ∈ S_WRITE then
22:            return true
23:        end if
24:    end for
25:    for all s ∈ GETWRITESYMBOLS(seg_2) do
26:        id ← id_2
27:        if ISCHANNELSYMBOL(s) then
28:            id ← TRANSLATECHANNELID(id_2, GETPCP(s))
29:        end if
30:        if id ∈ S_READ ∨ id ∈ S_WRITE then
31:            return true
32:        end if
33:    end for
34:    return false
35: end function
```



(a) Channel Segments (b) Linking (Old) (c) Cloning (New)

Fig. 9: Handling of channel segments from Fig. 6

Our new segment graph generation with the PCP extension uses a more sophisticated strategy. Specifically, each expression in a channel takes into account the individual port call context for the analysis, like `ch2++` in Fig. 9a. So, sharing of pre-analyzed channel segments between port calls is not possible. Similar to the regular segment graph, we analyze all channel functions first. Next, we continue with the analysis of the simulation threads, but when we hit a port call `p`, the related segment graph for the channel function is *cloned* and inserted instead of linked. Therefore, channel related segments

appear for each port call individually as in Fig. 9c. Concluding, the PCP of the cloned segments is extended. First, the PCP of the port call `p` is prepended to the PCP of the cloned segments. Second, the port of `p` itself is added to the cloned segments. This strategy is needed for the analysis of nested port calls in hierarchical channels.

D. Event Notification Table

The event notification table contains the notification dependencies between the individual segments. This information is needed for the simulator to schedule threads with respect to possible wake-up times. Without this information, threads could run ahead, events get lost, and the simulation fails. Since event notifications are analyzed with the same approach as regular variables, it is important to take the instance ID of events into account. So that false positives do not create extraneous conflicts, we apply the new technique of the PCP analysis for the event notification analysis as well.

IV. Experiments

We have implemented the Port Call Path analysis to demonstrate the importance for parallel simulation. The following experiments show the reduction of false positive data conflicts and event notifications. Additionally, we show the speedup between the sequential and the parallel execution with (pcp) and without Port Call Path (old). Our experiments consist of four different application examples, namely a video Mandelbrot renderer, a high-level video decoder, a bitcoin miner, and a NoC particle simulator. The execution times are measured on an Intel Xeon E3-1240 processor with 4 cores at 3.4 GHz. To obtain unambiguous measurements, we have turned CPU frequency scaling and hyper-threading off.

Note that in contrast to [9], we had to create for each individual channel instance an individual channel class resulting in large code duplication. Otherwise, the conflicting channel variables caused too many FP conflicts. In this work, all models contain only a single channel class with multiple instances.

A. Video Player

In the video player example, the stimulus sends data to an audio and a video decoder. After decoding, the processed stream goes to two speakers and one display unit. The design uses a generic channel type to transfer the data between the units. Without the PCP analysis, the individual channels are blocking each other. Additionally, they prevent the parallel decoding. In turn, the execution is effectively sequential and has the same execution time as the sequential reference simulation. In contrast, the new PCP analysis largely reduces the channel related false positive conflicts. Consequently, communication and computation can run in parallel. This enables a speedup of 1.48x (from 20.36 sec down to 13.76 sec) while reducing the data conflicts from 484 to 196 and the event notifications from 168 to 36.

B. Mandelbrot Renderer

The Mandelbrot renderer is a parallel video application to compute the Mandelbrot set. Fig. 10 shows the architecture of the model. Basically, the controller orchestrates a number of renderer units. Each unit computes a different slice of the image. During the simulation, the controller triggers the individual slices. Then, a slice computes the Mandelbrot set for given coordinates in a shared memory. The controller receives a completion signal from the units and then saves the frame. Finally, new coordinates are provided to all slices for the next frame.

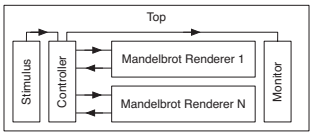


Fig. 10: Structure of a Mandelbrot video renderer

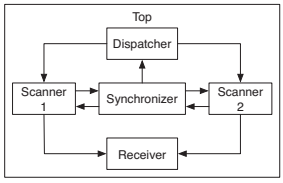


Fig. 11: Structure of a bitcoin miner model [14]

Table I shows the experiments of the Mandelbrot renderer application with up to 8 computation units. For all configurations, the analysis is performed with (pcp) and without (old) the PCP technique. The new analysis eliminated 2,024 false positive conflicts for the set with 8 units. The false positive conflicts effectively sequentialize the parallel execution. The PCP technique enabled a speedup of up to 3.36x on the 4-core host.

TABLE I: Speedup of the Mandelbrot video renderer

Renderer Units	1.old	1.pcp	2.old	2.pcp	4.old	4.pcp	8.old	8.pcp
Entries in table	441	441	784	784	1,764	1,764	4,900	4,900
Data conflicts	162	154	270	190	702	262	2,430	406
Event notifications	73	40	146	58	424	94	1,508	166
Speedup	1.00	0.99	1.00	1.68	1.00	2.90	1.00	3.36

C. Bitcoin Miner

Bitcoin is a digital currency and is established as a decentralized payment system [14]. Bitcoin mining describes the process of generating new bitcoins through solving math problems. When a new bitcoin enters the system it must prove the correctness of creation. Particularly, this proof happens through a cryptographic hash algorithm. The bitcoin miner application is doing this with three stages which include work dispatching, scanning, and result receiving. Fig. 11 shows the basic structure of the SystemC model. The scanners compute the hash algorithms for a given coin in parallel. If a scanner completes the computation of a coin, it synchronizes with the dispatcher and a new coin is sent to the scanners.

TABLE III: Speedup of the bitcoin miner example

Worker	1.old	1.pcp	2.old	2.pcp	4.old	4.pcp	8.old	8.pcp
Entries in table	100	100	256	256	784	784	2704	2704
Data conflicts	84	84	209	129	641	237	2069	501
Event notifications	4	4	8	4	16	4	32	4
Speedup	1	1	1	1.97	0.99	3.56	0.99	3.5

Table III shows the results for the bitcoin miner experiment. The design was executed with 1, 2, 4, and 8 scanners. The amount of false positive entries effectively sequentializes the execution of the old parallel version. In comparison, the new

TABLE II: Reduced false positive conflicts and speedup through the PCP analysis of the NoC particle simulator

	2x2.old	2x2.pcp	3x3.old	3x3.pcp	4x4.old	4x4.pcp	5x5.old	5x5.pcp	6x6.old	6x6.pcp
Number of Conflict Table Entries	14,641	14,641	73,441	73,441	231,361	231,361	564,001	564,001	1,168,561	1,168,561
Data conflicts (DC)	14,641	1,999	73,441	4,621	231,361	8,303	564,001	13,181	1,168,561	19,363
Event notifications (EN)	8,988	843	28,153	1,893	88,465	3,363	348,600	5,253	722,556	7,563
False positive DC in number / percent	12,642	86.35%	68,820	93.71%	223,058	96.41%	550,820	97.66%	1,149,198	98.34%
False positive EN in number / percent	8,145	90.62%	26,260	93.28%	85,102	96.20%	343,347	98.49%	714,993	98.95%
Execution time seq. in sec	2,354.24		1,048.78		590.37		378.79		262.58	
Execution time par. in sec	2355.7	1191.52	1046.1	577.66	590.57	195.82	379.98	134.87	270.31	86.96
Relative Speedup	1.00	1.98	1.00	1.82	1.00	3.01	1.00	2.81	0.97	3.02

PCP analysis resolved the conflicts. As a result, the individual workers execute in parallel and gain a speedup of up to 3.5x.

D. Network-on-Chip Particle Simulator

Our last experiment simulates a Network-on-Chip (NoC) particle simulator to demonstrate the importance of reduced false positive table entries. A tile communicates with its neighbors through bidirectional channel to the north, south, east, and west. In this model, particles move in a 2-dimensional space and affect each other. Each tile covers the computation of the moving particles for are defined area in the space. If a particle leaves the space of a tile, it transitions to the neighbor tile through channel communication.

The simulation of the particle simulator has three major stages. In the first stage, the platform communicates a set of particles to the individual tiles as an initial configuration. In the following second stage, the simulation of the particles starts and tiles compute the position for their associated particles. Particles are exchanged between neighboring tiles if they leave the scope of their hosting module. In the final step, the tiles communicate the position of the hosting particles back to the platform.

Table II shows the results of the particle simulator for grid sizes from 2x2 up to 6x6. For each size, the model is analyzed with (pcp) and without (old) the PCP analysis.

The first important observation is that both conflict tables have the same amount of entries for a given grid size.

Second, the amount of conflicts varies between the two versions. On one hand, in the old version all entries in the data conflict (DC) table are marked as conflict. In other words, all segments are conflicting with each other. Consequently, a simulation is effectively sequential. On the other hand, in the pcp version not all table entries are marked as conflict. For the 2x2 example the data conflict table has 14,641 (old) and 1,999 (pcp) positive conflict entries. This means that the old analysis caused 12,642 false positive conflicts which are 86.53% of the table entries. With increasing grid size, the amount of false positive conflicts increases regarding to the number of channels. The channel related false positive errors is increasing up to 98% for the 6x6 grid size. A similar observation can be made for the event notification (EN) table.

Third, the speedup for the old version is 1 at most. This means that parallel simulation had no impact. In contrast, the PCP sensitive analysis has a speedup of up to 3x.

V. Conclusion and Future Work

In this paper, we extended a fully automatic compiler infrastructure to parallelize IEEE SystemC simulation. Our new Port Call Path technique eliminates false positive conflicts in the analysis which impacts especially channel communication. Before, false conflicts severely compromised the parallel simulation of channels as well the computation in modules. We demonstrated the importance of the PCP technique for diverse examples and reduced the amount of false conflicts by up to 98%. As a result, we gained a speedup of up to 3.5x on a 4-core host machine.

In future work, we plan to extend our technique to SystemC TLM 2.0. Here, modules are allowed to modify the data of other modules without a linking channel, making he analysis even more difficult.

ACKNOWLEDGMENT

This work has been supported in part by substantial funding from Intel Corporation for the project titled "Out-of-Order Parallel Simulation of SystemC Virtual Platforms on Many-Core Architectures". The authors thank Intel Corporation for the valuable support.

References

[1] "IEEE Standard SystemC Language Reference Manual, IEEE Std 1666-2011," 2011.
[2] G. Glaser, G. Nitschey, and E. Hennig, "Temporal Decoupling with Error-Bounded Predictive Quantum Control," in *FDL*, 2015.
[3] N. Ventroux and T. Sassolas, "A New Parallel SystemC Kernel Leveraging Manycore Architectures," in *DATE*, 2016.
[4] J. H. Weinstock, R. Leupers, G. Ascheid, D. Petras, and A. Hoffmann, "SystemC-Link: Parallel SystemC Simulation using Time-Decoupled Segments," in *DATE*, 2016.
[5] G. Liu, T. Schmidt, and R. Dömer, "RISC Compiler and Simulator, Release V0.4.0: Out-of-Order Parallel Simulatable SystemC Subset," 2017.
[6] R. M. Fujimoto, "Parallel Discrete Event Simulation," *Commun. ACM*, vol. 33, no. 10, 1990.
[7] W. Chen, X. Han, and R. Dömer, "Out-of-Order Parallel Simulation for ESL Design," in *DATE*, 2012.
[8] W. Chen and R. Dömer, "An Optimizing Compiler for Out-of-Order Parallel ESL Simulation Exploiting Instance Isolation," in *ASPDAC*, 2012.
[9] T. Schmidt, G. Liu, and R. Dömer, "Exploiting Thread and Data Level Parallelism for Ultimate Parallel Systemc Simulation," in *DAC*, 2017.
[10] T. Schmidt, G. Liu, and R. Dömer, "Hybrid analysis of systemc models for fast and accurate parallel simulation," in *ASPDAC*, 2017.
[11] J. H. Weinstock, C. Schumacher, R. Leupers, G. Ascheid, and L. Tosoratto, "Time-Decoupled Parallel SystemC Simulation," in *DATE*, 2014.
[12] R. Sinha, A. Prakash, and H. D. Patel, "Parallel Simulation of Mixed-abstraction SystemC Models on GPUs and Multicore CPUs," in *ASP-DAC*, 2012.
[13] C. Roth, S. Reder, H. Bucher, O. Sander, and J. Becker, "Adaptive algorithm and tool flow for accelerating systemc on many-core architectures," in *Digital System Design (DSD), 17th Euromicro Conference*, 2014.
[14] Z. Cheng and R. Dömer, "A SystemC Model of a Bitcoin Miner," in *Technical Report CECS-16-04*, 2016.