Analyzing Variable Entanglement for Parallel Simulation of SystemC TLM-2.0 Models

ZHONGQI CHENG and RAINER DÖMER, University of California, Irvine

The SystemC TLM-2.0 standard is widely used in modern electronic system level design for better interoperability and higher simulation speed. However, TLM-2.0 has been identified as an obstacle for *parallel* SystemC simulation due to the disappearance of channels. Without a containment construct, simulation threads are permitted to directly access data of other modules and that makes it difficult to synchronize such accesses as required by the SystemC execution semantics. In this paper, we propose a compile time approach to statically analyze potential conflicts among threads in SystemC TLM-2.0 loosely- and approximately-timed models. We introduce a new Socket Call Path technique which provides the compiler with socket binding information for precise static analysis. We also propose an algorithm to analyze entangled variable pairs. Experimental results show that our approach is able to support automatically safe parallel simulation of SystemC models with TLM-2.0 Blocking Transport Interface, Direct Memory Interface and Non-blocking Transport Interface, resulting in impressive simulation speeds.

CCS Concepts: • **General and reference** → **Validation**; **Verification**;

Additional Key Words and Phrases: SystemC, TLM-2.0, parallel discrete event simulation, PDES

ACM Reference format:
Zhongqi Cheng and Rainer Dömer. 2019. Analyzing Variable Entanglement for Parallel Simulation of SystemC TLM-2.0 Models. *ACM Trans. Embed. Comput. Syst.* 18, 5s, Article 79 (October 2019), 20 pages.
https://doi.org/10.1145/3358194

1 INTRODUCTION

The Open SystemC Initiative (OSCI) TLM-2.0 [1] is a transaction level modeling standard released in 2008. The typical use of it is building virtual platforms to simulate today's large system on chip (SOC) models with processors, buss and other components. TLM-2.0 consists of a set of core interfaces, sockets, generic payload and so on. These facilities are used to increase (a) interoperability between models, that is, the plug-and-play ability to take transaction level models from different sources and connect them together and (b) simulation speed.

Out-of-Order Parallel Discrete Event Simulation (OoO PDES) [2] has been designed for highly parallel SystemC simulation. Compared to traditional PDES, OoO PDES utilizes a finer grained task scheduler to allow suitable threads running in parallel even when they are in different cycles.

This article appears as part of the ESWEEK-TECS special issue and was presented in the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2019.
Authors' address: Z. Cheng and Rainer Dömer, 3217 Engineering Hall, University of California, Irvine, The Henry Samueli School of Engineering, Electrical Engineering & Computer Science, Irvine, CA 92697-2625, USA; emails: {zhongqc, doemer}@uci.edu.

1539-9087/2019/10-ART79 $15.00
https://doi.org/10.1145/3358194

This significantly increases the execution speed. In order to preserve the simulation semantics and timing accuracy, the input model is statically analyzed by a dedicated compiler to prevent potential race conditions between threads [2].

Conceptually, TLM-2.0 and OoO PDES are both library-based approaches and can be applied together. However, no analysis technique has been designed until today that is capable of determining safe parallelism between threads in SystemC TLM-2.0 models.

In this paper, we propose and implement a static analysis for TLM-2.0 loosely-timed (LT) and approximately-timed (AT) SystemC models with multiple threads. Our approach is able to precisely and automatically analyze potential conflicts between threads that are communicating using the standard TLM-2.0 interfaces, and helps the model to achieve reasonable execution speedup given its parallelism potential. In this paper, we describe in detail the analysis of the two mostly used APIs, namely the Blocking Transport Interface (BTI) and Direct Memory Interface (DMI). These two interfaces use the main TLM-2.0 features, such as generic payload, DMI objects and sockets. We also outline the support of the Non-Blocking Transport Interface (NBTI) and Debug Transport Interface (DTI) using the same techniques proposed in this paper. Note that Register-Transfer Level (RTL) and TLM-1.0 models that do not use sockets are not targeted in this work. Given the results of the novel static analysis, a SystemC TLM-2.0 model can then be simulated safely and fast with OoO PDES, as demonstrated with extensive experiments and results in Section 8.

1.1 Motivation

OoO PDES has shown excellent results for faster simulation of SystemC TLM-1.0 based models. However, as pointed out in [3], TLM-2.0 is an obstacle for parallel simulation. In contrast to TLM-1.0, TLM-2.0 lacks the concept of channels. Instead, a module uses *pointers* to access memory locations in other modules. Since pointer analysis is difficult and communications are not encapsulated in containment constructs, no multi-thread access synchronization can be offered and race conditions are likely to occur, violating the execution semantics. [3] also proposed a conceptual solution for this problem, namely the idea of reintroducing channels into TLM-2.0 and protect the communication with locks. However, this would reduce simulator speed and violate interoperability of the TLM-2.0 IEEE standard [1].

In order to simulate TLM-2.0 models safely and correctly with OoO PDES, we propose a new static analysis-based approach that protects communication without the need to modify the TLM-2.0 standard nor the application model. Our technique builds on top of an analysis algorithm for SystemC TLM-1.0 that is implemented in the Recoding Infrastructure for SystemC (RISC) [4]. It is a compiler based approach that automatically analyzes data, timing and event hazards among threads. However, the TLM-1.0 oriented static analysis is not able to analyze *variable entanglement* and is thus insufficient for the TLM-2.0 standard.

Definition 1.1. Variable Entanglement: Variable Entanglement occurs when one variable points to the memory location of another variable in other modules through the SystemC TLM-2.0 communication interfaces. Because two entangled variables refer to the same memory location, access (read/write) to one is also applied to the other.

As an example, Figure 1 shows a SystemC TLM-2.0 model of a DVD player which decodes a stream of H.264 video and MP3 audio data using separate decoders. All communications are modeled using TLM-2.0 sockets and APIs. Three parallel threads `T_Video`, `T_Audio_Left` and `T_Audio_Right` in the initiator `stimulus` store data into corresponding target memories `mem1`, `mem2` and `mem3`. Decoders `vDecoder`, `alDecoder` and `arDecoder` fetch the data from the memories and decode it. In this example, there are three parallel lanes. Take the `stimulus-mem1-vDecoder` lane as an example. Through the TLM-2.0 interfaces, variable `vFrame` of `stimulus` and pointer `ptr`

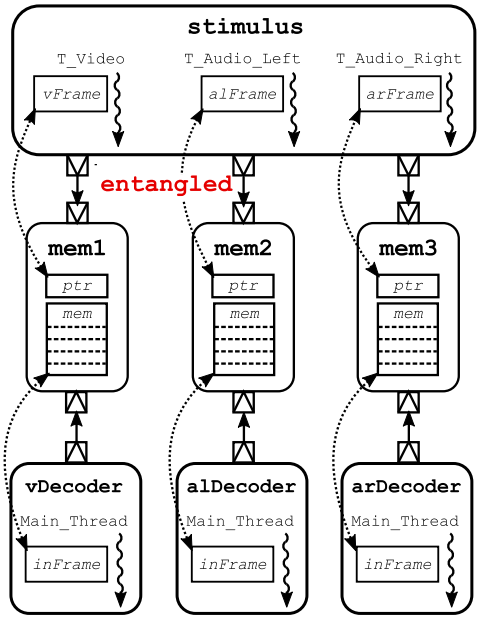


Fig. 1. SystemC TLM-2.0 model of a DVD player.

of `mem1` are entangled (indicated by the dashed arrow in Figure 1), meaning that the two variables are pointing to the same memory location. Similarly, `inFrame` of `vDecoder` and `mem` of `mem1` are entangled as well. Figure 2 lists partial code of this simple SystemC model and shows in detail how the variables are entangled.

Analyzing the variable entanglement information is critical for an accurate static analysis. It helps the SystemC compiler to handle the behavior of a pointer in a well-defined TLM-2.0 model. In general, statically analyzing the entanglement of general pointers is very difficult. However, we observe that the pointers for TLM-2.0 communication do not point to arbitrary memory locations. Through the well-defined TLM-2.0 interface methods, a pointer points to a determined memory location and thus can be analyzed statically. In the above example, for instance, `inFrame` of `vDecoder` points only to `mem` of `mem1` through the TLM-2.0 DMI interface. The prior TLM-1.0 oriented static analysis does not take this variable entanglement information into account and treats `inFrame` as a pointer potentially pointing to any memory location in the entire model. Consequently, thread `vDecoder::Main_Thread` that accesses `inFrame` is not allowed to run in parallel with any other threads to prevent race conditions. Such too-strict pointer-involved data conflict analysis causes false conflicts and may reduce the simulator speed to sequential levels. Now by analyzing variable entanglement, the compiler is able to identify that `inFrame` of `vDecoder` and `mem` of `mem1` are pointing to the same memory location, accessing `inFrame` only causes data hazard with the concurrent access to `mem`. With the precise and correct data conflict analysis, the three lanes in the example in Figure 1 are able to run in parallel with no race conditions, rather than sequentially due to pointer conflicts imposed by the TLM-1.0 oriented static analysis.

The proposed approach in this paper allows the SystemC compiler to analyze TLM-2.0 interfaces and socket connections. Based on this information, our approach then precisely analyzes the entangled variables. Note that our work does not support the analysis of general pointer operations, which is known to be hard for static analysis.

```
SC_MODULE(Stimulus){
  void T_Video(){
    tlm::tlm_generic_payload rgp;
    rgp.set_data_ptr(vFrame);
    ...//initilize rgp
    sc_time delay;
    while(true){
    //keep sending vFrame
      ...//generate vFrame
      out1->b_transport(rgp, delay);
      ...
    }
  }
  tlm_utils::simple_initiator_socket
    out1; //socket
  unsigned char vFrame[FRAMESIZE];
  ...//other functions and variables
};
```

```
SC_MODULE(VideoDecoder){
  void Main_Thread(){
    unsigned char inFrame[FRAMESIZE];
    tlm::tlm_generic_payload rgp;
    tlm::tlm_dmi rd;
    ... // initialize rgp, rd
    bool DMI_allowed =
      in->get_direct_mem_ptr(rgp, rd);
    inFrame = rd.get_dmi_ptr();
    while(DMI_allowed){
    //keep decoding inFrame
      decode(inFrame);
    }
  }
  tlm_utils::simple_initiator_socket
    in; //socket
  ...//other functions and variables
};
```

```
SC_MODULE(Memory){
  void custom_b_transport(tlm::tlm_generic_payload &pgp, sc_time &delay)
  {
    unsigned char *ptr = pgp.get_data_ptr();
    unsigned int len = pgp.get_data_length();
    ... //get other pgp fields
    memcpy(&mem[OFFSET], ptr, len);
    ... //error handling, response
  }
  bool custom_get_dmi(tlm::tlm_generic_payload& pgp, tlm::tlm_dmi& pd)
  {
    pd.allow_read_write();
    pd.set_dmi_ptr(&mem[OFFSET]);
    ... //set other pd fields and pgp fields
    return true;
  }
  Memory(){
    in.register_b_transport(this, &Memory::custom_b_transport);
    out.register_get_direct_mem_ptr(this, &Memory::custom_get_dmi);
  }
  //sockets and data
  tlm_utils::simple_target_socket in;
  tlm_utils::simple_target_socket out;
  unsigned char mem[SIZE];
  ...
};
```

Fig. 2. Partial SystemC code for Figure 1.

1.2 Related Work

Parallel Discrete Event Simulation (PDES) was first studied in [5]. In [2], Out-of-Order Parallel Discrete Event Simulation (OoO PDES) was proposed to further increase the simulation speed. Static analysis is an integral part of OoO PDES. It is performed by a SystemC-aware compiler to analyze the potential conflicts and race conditions within a system model. Here, a Segment Graph (SG) data structure represents the concurrent behavior of simulation threads. Also, an instance ID technique is introduced to augment the SG for more precise conflict analysis. Port call path [6] is another advanced technique that helps the compiler to gain more specific context information

about non-pointer variables in channels. This can reduce false positive conflicts in channels and result in significantly increased simulation speed.

The work described in this paper is inspired by [6] but different from it in several aspects. First, our Socket Call Path (SCP) technique is novel as it enables the compiler to analyze entangled pointers through TLM-2.0 interface APIs, which is not done in [6] (nor has been done in any other work to the best of our knowledge). Second, [6] focuses on the analysis of SystemC TLM-1.0 channels, whereas our work targets SystemC TLM-2.0. Note that TLM-1.0 and TLM-2.0 are entirely different in their communication modeling, because the channel concept has disappeared [3].

In [7], the authors achieve a high simulation speed of SystemC models by exploiting data-level parallelization together with thread-level parallelization. An algorithm was proposed to automatically apply data-level parallelization to the source code. This is orthogonal to the focus of this work and thus could be applied here as well.

TLM-2.0 was first identified as an obstacle for parallel SystemC simulation in [3]. Inter-thread communication is considered unsafe in a multi-thread environment when not encapsulated in a channel. To overcome the obstacle, the author proposed a conceptual solution that wraps the TLM-2.0 communication methods into actual channels (similar to TLM-1.0), so that locks could be implemented for protection. In contrast, our approach does not require extra communication containment nor locks. We protect simulation semantics by precise static analysis only.

Parallel SystemC simulation is also studied in many other works. In [8] SystemC-clang is proposed. It analyzes SystemC models with a mixture of transaction-level and register-transfer level components. In [9] the authors studied the distributed parallel simulation, where SystemC models are organized into small executable units and distributed onto different host machines to run in parallel. [10] provides a parallel SystemC simulation kernel which requires the user to manually translate the sequential design into a safe parallel design. [11] takes a survey about existing SystemC simulation approaches and concludes that most of these works do not fully support the parallel simulation of TLM-2.0 LT models due to shared variables. Our work addresses this identified problem. [12] proposes a tool that also addresses this problem with an alternative approach. A set of primitives is provided to the user to manually express tasks with duration such that parallelism in the model can be exploited and LT models are executing in parallel. In [13], a parallel SystemC simulation kernel is developed by reducing synchronization overheads of parallel threads. None of these works supports the safe and automatic parallel simulation of TLM-2.0 LT and AT models.

2 BACKGROUND

In this section, we first briefly review the TLM-2.0 interfaces [14, 15] and then describe the Segment Graph (SG) data structure utilized by OoO PDES.

2.1 TLM-2.0 Background

This section briefly reviews the usage of TLM-2.0 BTI and DMI with a partial code that describes the `stimulus-mem1-vDecoder` lane in Figure 1. The code snippet is shown in Figure 2. In the code, `Stimulus` is the module type of `stimulus`, `Memory` is the module type of `mem1` and `VideoDecoder` is the module type of `vDecoder`. TLM-2.0 focuses mainly on the communication between processes rather than computation, so the details of encoding and decoding algorithms are not shown in the code.

2.1.1 BTI. We first examine the Blocking Transport Interface (BTI). It is used for the communication between `stimulus` and `mem1` in Figure 1. Basically, it takes four steps to use BTI. On the initiator's side:

(1) prepare a generic payload object and a timing annotation. In this example, rgp is defined in line 3 and delay is defined in line 6. rgp stands for *Referred Generic Payload* and will be described in Section 3.3.2. In line 4, the video frame vFrame is wrapped inside rgp by set_data_ptr. Note that initializations of other rgp's fields (such as data length, streaming width and so on) are omitted in this demo code.
(2) call b_transport via the initiator socket, passing the prepared generic payload object and timing annotation as the arguments. In this example, b_transport is called on socket Stimulus::out1 with rgp and delay as arguments, in line 10.

On the target's side:

(1) implement a callback for b_transport. In this example, we implement Memory::custom_b_transport in line 19 as the callback method. The callback method takes a reference of generic payload pgp and the timing annotation delay as parameters. pgp stands for *Parametric Generic Payload* and will be described in Section 3.3.2. In line 21, the pointer wrapped inside pgp is extracted by get_data_ptr and assigned to ptr. The data length is extracted in line 22. Other extractions of pgp's fields are omitted in this demo code. Line 24 copies ptr's value to Memory::mem at offset OFFSET.
(2) register the callback on the target socket. In the Memory's constructor, Memory::custom_b_transport is registered as the b_transport callback method on socket Memory::in, in line 35.

A corresponding callback is executed on every b_transport call[1]. Consequently, the actual behavior of a b_transport is fully defined by its callback function. In the stimulus-mem1-vDecoder lane, stimulus is connected to mem1 via socket binding stimulus.out1→mem1.in. When stimulus.

T_Video executes out1->b_transport, it invokes mem1.custom_b_transport. In this case, the behavior of this b_transport is represented by Memory::custom_b_transport.

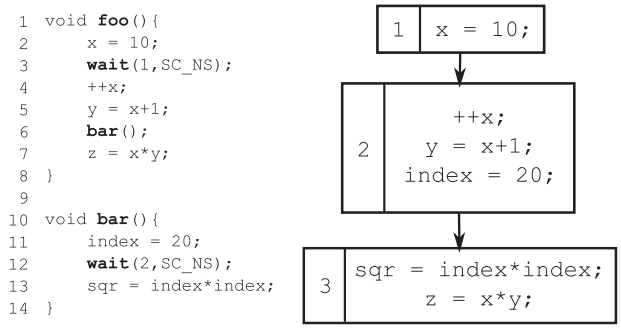
2.1.2 DMI. The Direct Memory Interface (DMI) is used for the communication between vDecoder and mem1 in Figure 1. Similar to BTI, it takes five steps to use DMI. On the initiator's side,

(1) prepare a generic payload object and a dmi object. In this example, thread VideoDecoder::Main_Thread declares rgp at line 47 and rd at line 48. rd stands for *Referred DMI object* and will be described in Section 4.
(2) call get_direct_mem_ptr via the initiator socket, passing the prepared generic payload object and DMI object as the arguments. In this example, get_direct_mem_ptr is called on socket VideoDecoder::in with rgp and rd as arguments, in line 50. The *Boolean* flag DMI_allowed indicates if the DMI access is permitted by the target.
(3) extract the pointer of the directly accessed memory location from the dmi object via get_dmi_ptr. In line 51, inFrame is the extracted pointer from rd, through which videoDecoder::Main_Thread is able to access the memory location in the connected target module instance directly.

On the target's side:

(1) implement a callback for get_direct_mem_ptr. In this example, we implement Memory::

[1]Section 3.1 explains how we match a b_transport call to the registered callback method(s).



(a) Source code (b) Segment Graph

Fig. 3. Example SystemC code and corresponding SG.

`custom_get_dmi` in line 27. The callback method takes a reference of generic payload `pgp` and a reference of dmi object `pd` as parameters. `pd` stands for *Parametric DMI object* and will be described in Section 4. In line 30, the pointer to `Memory::mem` at offset `OFFSET` is wrapped inside `pd` by `set_dmi_ptr`. Other initializations of `pd`'s fields are omitted in this demo code.

(2) register the DMI callback on the target socket. In the `Memory` constructor, `Memory::custom_get_dmi` is registered as the DMI callback method on socket `Memory::out`, in line 36.

In the `stimulus-mem1-vDecoder` lane, `mem1` and `vdecoder` are connected via socket binding `vDecoder.in→mem1.out`. Through `get_direct_mem_ptr`, `vDecoder` gets the direct memory access to `mem1.mem`.

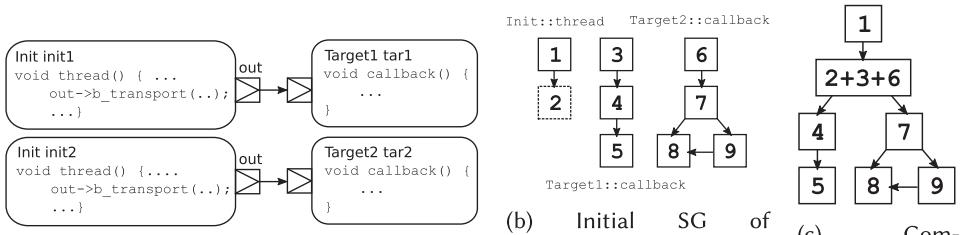
2.2 Segment Graph and Function Call Analysis

The SG is a directed graph where each node is a set of code statements executed between two scheduling steps [2]. A scheduling step is the entry to the scheduler domain from the application domain during execution of the model, which includes `wait` statement, start of a thread and end of a thread. An example of SystemC source code is shown in Figure 3(a). The corresponding SG is shown in Figure 3(b).

A SG is automatically built by the SystemC-aware compiler as discussed in Algorithm 5 in [16]. The algorithm analyzes every source code statement and groups them into segments. For instance, `++x` in line 4 and `y=x+1` in line 5 are assigned both to segment 2 as they are executed after the same scheduling step: `wait(1,SC_NS)` in line 3. Function call is a special case because it may contain multiple statements and/or scheduling steps. When a function call is encountered, the algorithm first finds the definition of the function and constructs a temporary SG *sg_func* for the function. Then, *sg_func* is merged into the calling segment. In this example, `bar()` is first encountered while building segment 2. The RISC compiler identifies the definition of `bar()` in the Abstract Syntax Tree (AST) and builds a SG *sg_func* for `bar()`. Then, *sg_func* is merged into segment 2.

3 STATIC ANALYSIS FOR BLOCKING TRANSPORT INTERFACE

The BTI is appropriate when an initiator wants to complete a transaction with a target during the course of a single function call, to be specific, `b_transport`. In this section, we first discuss the approach to build the needed SG for `b_transport` call. Then, we introduce our new Socket Call



(a) SystemC model with two initiators and two targets

(b) Initial SG of `Init::thread` and temporary SGs of callback methods

(c) Complete SG of `Init::thread` after merging

Fig. 4. Example of merged SGs of multiple callbacks.

Path (SCP) technique that provides context information about socket bindings. Finally, we propose an approach for analyzing variable entanglement.

3.1 Segment Graph for `b_transport`

The TLM-2.0 BTI is intended to support the loosely-timed coding style. The source code in Figure 2 demonstrates the use of BTI. As discussed in Section 2.1, the behavior of a `b_transport` is represented by its registered callback method on the target's side. Thus, to build the SG for a `b_transport` call, the RISC compiler must be able to find the function definition of the corresponding callback method. In our proposed approach, this is done in three steps:

(1) for each target socket, identify the registered `b_transport` callback method.
(2) use a recursive approach to obtain the socket binding information. The approach is recursive because an initiator socket may bind to another intermediate initiator/target socket, and we need to get an end-to-end binding information.
(3) when a `b_transport` call is encountered in a calling segment, the compiler first identifies the bound target socket according to the socket binding information, then builds the SG of the callback method registered on this target socket. Finally, the compiler merges the SG into the calling segment.

Note that a single `b_transport` can potentially have multiple callback methods. An example is shown in Figure 4(a). In this example, `init1` and `init2` are module instances of the same module type `Init`, whereas `tar1` and `tar2` are of type `Target1` and `Target2` respectively. When `out->b_transport` is executed in `Init::thread`, either `Target1::callback` or `Target2::callback` may be called depending on which `Init` module instance the `b_transport` belongs to. Note that an SG represents threads at module level but not instance level. This means that when building the SG for `Init::thread`, the compiler cannot determine if the current `Init::thread` is called from module instance `init1` or `init2`. Thus, our proposed static analysis takes this flexible binding behavior into account. We illustrate processing of callback methods in Figure 4(b) and 4(c). Initially, `Init::thread` has two segments: segment 1 and 2. Segment 2 is being constructed (depicted as dashed border) and here `out->b_transport` is called. Then, the compiler builds temporary SGs for `Target1::callback` and `Target2::callback`, where segments 3, 4 and 5 belong to `Target1::callback` and segments 6, 7, 8, and 9 belong to `Target2::callback`. Since `Target1::callback` and `Target2::callback` are both potential callbacks of `out->b_transport`, their temporary SGs are joined into the complete SG of `Init::thread` in Figure 4(c). Specifically, segment 3 and segment 6 are both merged with segment 2, represented by 2+3+6 in Figure 4(c).

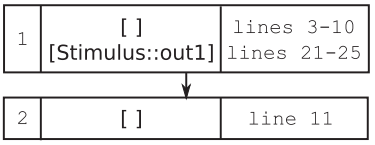


Fig. 5. SCP-included SG of `Stimulus::T_Video` in Figure 2.

3.2 Socket Call Path

Socket Call Path (SCP) is a new advanced technique in our approach. It provides the SystemC compiler with the information regarding how a target is reached by the initiator through the TLM-2.0 interface. The idea is similar to the Port Call Path (PCP) analysis proposed in [6]. One main difference is that PCP is based on port-to-channel connections whereas SCP is for analyzing socket-to-module connections. Also different from PCP, a SCP is represented by a list of sockets. When used together with SG, SCP helps the SystemC compiler to perform instance-aware conflict analysis, which provides similar benefits as to the use of PCP in [6]. Figure 5 shows a SCP-included SG of `Stimulus::T_Video` in Figure 2. Segment 1 contains code statements in lines 3–10 from `Stimulus::T_Video` and lines 21–25 from the callback method `Memory::custom_b_transport`. On one hand, lines 3–10 are local to `Stimulus::T_Video` and thus have an empty SCP. On the other hand, lines 21–25 are accessed through socket `Stimulus::out1` and their corresponding SCP is [Stimulus::out1]. Given the SCP information, the SystemC compiler looks up the socket binding information and identifies that when lines 21–25 are executed by thread `T_Video` of `stimulus`, they belong to `mem1`. The SCP-included SG captures an instance-aware behavior of threads and thus increases the precision of conflict analysis.

3.3 Variable Entanglement Analysis

Variable entanglement is defined in Definition 1.1. Figure 1 in Section 1.1 shows an example of variable entanglement through TLM-2.0 interfaces. Without precise analysis, entangled variables result in overwhelming false conflicts and thus compromise the simulation. In this section, we propose a three-step approach to analyze variable entanglement and variable access for entangled variables.

3.3.1 Identify Original and Alias Variable. In this step, the compiler identifies (a) the original variable encapsulated in a generic payload by `set_data_ptr`, and (b) the alias variable extracted from a generic payload by `get_data_ptr`. In Figure 1, `vFrame` in line 4 is an original variable and `ptr` in line 21 is an alias variable. The original and alias variables are stored in a table data structure for later use.

3.3.2 Reference Analysis for Generic Payload with SCP. In the second step, the compiler analyzes the mapping between *parametric generic payload* (PGP) and *referred generic payload* (RGP). PGP is a reference parameter of a `b_transport` callback method, for instance, `pgp` in line 19 of Figure 2. RGP is the generic payload passed to a `b_transport` call, for instance, `rgp` in line 3 of Figure 2. A PGP refers to RGP(s) through BTI.

Because a `b_transport` callback method may serve as the callback of multiple `b_transport` calls from various initiators, its PGP can potentially refer to multiple RGPs. In the example in Figure 6, `Target::callback()` is the callback method for both `b_transport` calls in `Init1::thread` and `Init2::thread`. At runtime, `pgp` of `Target::callback()` refers at different times to the two RGPs: `Init1::rgp` and `Init2::rgp`. However, the static compiler is not able to further identify the exact RGP that a PGP refers to in a given context. For instance, the compiler cannot figure out that `pgp` actually refers to `Init1::rgp` but not `Init2::rgp` when

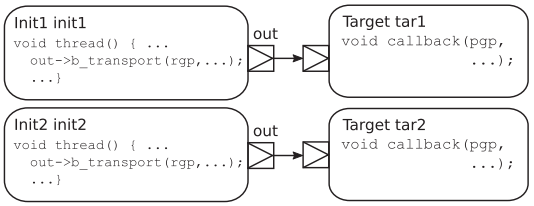


Fig. 6. SystemC model with two initiators and two targets.

`Target::callback` is invoked by the `b_transport` in `Init1::thread`. Such ambiguity causes many false conflicts if a PGP refers to a large number of RGPs.

To solve this problem, we attach SCP as a context information to each RGP. As described in Section 3.2, a SCP represents the path from a `b_transport` call to its callback method. Since RGP is the argument of a `b_transport` call and PGP is the parameter of the callback method, the SCP also represents the connection between RGP and PGP. For the example in Figure 6, when `Target::callback` is invoked by the `b_transport` in `Init1::thread`, `pgp` refers to `Init1::rgp` via SCP [Init1::out]. On the other hand, when `Target::callback` is invoked by the `b_transport` in `Init2::thread`, `pgp` refers to `Init2::rgp` via SCP [Init2::out]. The SCP-included PGP-RGP reference mappings are stored in a table data structure for later use.

3.3.3 Variable Access Analysis for Entangled Variables. Through the PGP-RGP reference mappings, the corresponding alias and original variables are entangled. Algorithm 1 shows the algorithm to analyze variable accesses for entangled variables within a segment. It is divided into two parts. First, two sets $\mathbb{R}$ and $\mathbb{W}$ are created in lines 3–5 to store read and written variables in the segment. In particular, we traverse each expression of the segment and identify the accessed variables with AnalyzeExpression. Second, in lines 8–33, the algorithm checks for each accessed variable if it has an entangled variable. If so, the entangled variable is also added to the corresponding set.

Take segment 1 in Figure 5 and Figure 2 as an example. When expression `memcpy(&mem[OFFSET], ptr,len)` in line 24 is encountered, Algorithm 1 first analyzes the variables accessed in this expression and assigns them to corresponding variable access sets. In this example, `mem` with SCP [Stimulus::out1] is assigned to $\mathbb{W}$, `ptr` with SCP [Stimulus::out1] and `len` with SCP [Stimulus::out1] are assigned to $\mathbb{R}$. Next, Algorithm 1 identifies alias variables in the two sets according to the information collected in the step in Section 3.3.1. The only alias variable is `ptr` in $\mathbb{R}$. It is extracted from `pgp` in line 21 of Figure 2. A PGP can refer to multiple RGPs as described in the step in Section 3.3.2. We collect them all in a set $\mathbb{RGP}$. Since `ptr` is reached through SCP [Stimulus::out1], given this context information, `pgp` should refer to an RGP that has the same SCP, [Stimulus::out1]. In line 27 of Algorithm 1, such RGP is identified, which is `rgp` in `Stimulus::T_Video`. It encapsulates an original variable `vFrame`. Finally, `vFrame` is assigned to $\mathbb{R}$.

While a detailed theoretical complexity analysis is beyond the scope of this paper, we note that the size of the analysis tasks performed by the compiler is proportional to the model size. Specifically for Algorithm 1, all function calls (e.g., IsAliasVar, IdentifyPGPs) performed inside have constant time complexity[2]. Thus, the overall time complexity of Algorithm 1 is $O(N^2)$ due to the nested `for` loops, where N is the total number of read and written variables in the segment.

[2]Technically, these functions are implemented to look up hash tables that are constructed in previous steps.

ALGORITHM 1: Variable access analysis for entangled variables

```
1: function AnalyzeVariableAccess(seg)
2:     //build the variable read and write sets
3:     for all exprWithScp ∈ seg do
4:         AnalyzeExpression(exprWithScp, ℝ, 𝕎)
5:     end for
6:
7:     //add entangled variables to variable write set
8:     for all varWithScp ∈ 𝕎 do
9:         if IsAliasVar(varWithScp) then
10:            pgp ← IdentifyPGP(varWithScp)
11:            ℝ𝔾ℙ ← IdentifyRGPs(pgp)
12:            for all rgp ∈ ℝ𝔾ℙ do
13:                if GetSCP(rgp) == GetSCP(varWithScp) then
14:                    originalVar ← GetOriginalVar(rgp)
15:                    AddVariableAccess(originalVar, 𝕎)
16:                end if
17:            end for
18:        end if
19:    end for
20:
21:    //add entangled variables to variable read set
22:    for all varWithScp ∈ ℝ do
23:        if IsAliasVar(varWithScp) then
24:            pgp ← IdentifyPGP(varWithScp)
25:            ℝ𝔾ℙ ← IdentifyRGPs(pgp)
26:            for all rgp ∈ ℝ𝔾ℙ do
27:                if GetSCP(rgp) == GetSCP(varWithScp) then
28:                    originalVar ← GetOriginalVar(rgp)
29:                    AddVariableAccess(originalVar, ℝ)
30:                end if
31:            end for
32:        end if
33:    end for
34: end function
```

4 STATIC ANALYSIS FOR DIRECT MEMORY INTERFACE

In this section, we discuss the static analysis for TLM-2.0 DMI. It is consistent with the approach for BTI with a few key differences.

The TLM-2.0 DMI is designed to speed up simulation by giving an initiator a direct pointer to an area of memory in a target. It involves two directions of communication paths:

(1) *Forward path* lets the initiator request a direct memory pointer from a target with the `get_direct_mem_ptr` method call. In the example in Figure 1, `inFrame` of `vDecoder` is entangled with `mem` of `mem1` through the forward DMI path.
(2) *Backward path* lets the target invalidate a DMI pointer previously given to an initiator with the `invalidate_direct_mem_ptr` method call.

To support the DMI in the proposed approach, we only need to consider the forward path. The backward path is not analyzed because a static analysis must be conservative and consider

variable entanglement at all times, not only between get_direct_mem_ptr and invalidate_direct_mem_ptr function calls. Besides, the backward path does not entangle variables.

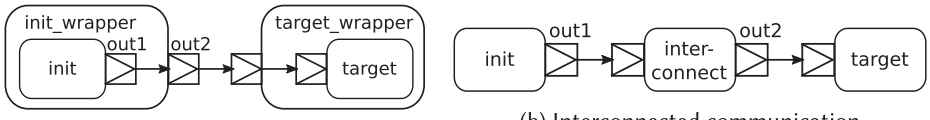
We utilize a similar approach as for BTI to statically analyze DMI. The approach involves two main steps:

(1) Construct SCP included SG for get_direct_mem_ptr call. The same approach as in Section 3.1 and 3.2 is used. The compiler first identifies the callback method of a get_direct_mem_ptr call using the socket binding information. Then, the SG of the callback is built and merged to the calling segment with a correct SCP.
(2) Analyze variable entanglement through DMI. A similar approach as the three steps in Section 3.3 is used. The SystemC compiler first analyzes the original variable wrapped inside a *parametric DMI object* (PD) and the alias variable extracted from a *referred DMI object* (RD). In Figure 2, Memory::mem is the original variable wrapped inside pd in line 30. inFrame of VideoDecoder:: Main_Thread is an alias variable extracted from rd in line 51. Next, the compiler analyzes the reference mapping between PD and RD. In Figure 1, vDecoder is connected to mem1. With this module interconnect information, the SystemC compiler identifies that in Figure 2, Memory::custom_get_dmi is the DMI callback method of get_direct_mem_ptr in line 50, and thus pd refers to rd via SCP [VideoDecoder::in]. Finally, the variable accesses for entangled variables are analyzed. From the previous two steps, the compiler obtains the information that pd wraps Memory::mem and rd releases inFrame, and pd refers to rd via SCP [VideoDecoder::in]. Consequently, inFrame is entangled with Memory::mem via SCP [VideoDecoder::in]. Since inFrame is read in line 54 in Figure 2, Memory::mem with SCP [VideoDecoder::in] is thus assigned to $\mathbb{R}$ of the corresponding segment.

5 STATIC ANALYSIS FOR NON-BLOCKING TRANSPORT INTERFACE

Non-blocking Transport Interface (NBTI) is intended to support the approximately-timed coding style. It breaks down a transaction into multiple timing points, and typically requires multiple function calls with phase information for a single transaction. Two interface methods nb_transport_fw and nb_transport_bw are used for forward and backward communications between initiators and targets. Similar to b_transport, the behaviors of nb_transport_fw and nb_transport_bw are represented by their corresponding registered callback methods.

nb_transport_fw and nb_transport_bw take three parameters: a generic payload, a timing annotation and a phase object. The generic payload and timing annotation are of the same use as for b_transport. The phase object is used to indicate the current phase of a transaction. Since a static analysis is conservative and considers variable entanglement at all times, not only between certain phases, transaction phase updates need not to be treated specially in a static analysis. With all the above observations, NBTI can be analyzed the same way as for BTI. The SystemC compiler first builds the SCP-included SG for nb_transport_fw and nb_transport_bw according to the registered callback methods and socket binding information. Then, it analyzes the variable entanglement through NBTI. BTI and NBTI both entangle variables through the PGP-RGP reference mappings, and thus variable entanglement by NBTI can be analyzed using the approach proposed in Section 3.3. One main difference between NBTI and BTI is that NBTI contains backward paths (form target sockets to initiator sockets) used by nb_transport_bw, whereas in BTI there are only forward paths (from initiator sockets to target sockets). To support the nb_transport_bw API, we augment our SystemC Internal Representation with backward socket mapping information.



(a) Hierarchical communication

(b) Interconnected communication

Fig. 7. Example of indirect communications.

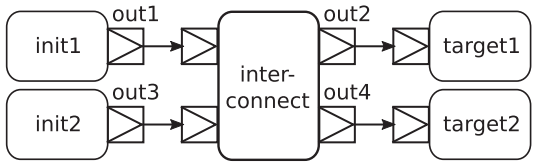


Fig. 8. Interconnected communication with multiple initiators and targets.

6 STATIC ANALYSIS FOR DEBUGGING TRANSPORT INTERFACE

Debugging Transport Interface (DTI) is used for debug access which gives an initiator the ability to read or write memory in the target without delays or side-effects. `transport_dbg` is the corresponding interface method and takes only a generic payload as parameter. Conceptually, DTI can be statically analyzed the same way as for BTI. SCP-included SG for `transport_dbg` is firstly built and variable entanglement through DTI is analyzed using the approach proposed in Section 3.3.

7 STATIC ANALYSIS FOR INDIRECT COMMUNICATION

In the previous sections, all examples involve only direct communication between initiators and targets. Nevertheless, our proposed solution also works for indirect TLM-2.0 communication, for instance, communication through hierarchical modules and interconnect components such as routers and bridges. Two corresponding examples are shown in Figure 7(a) and 7(b).

7.1 Hierarchical Communication

In Figure 7(a), an initiator `init` is connected to a target `target` across wrappers `init_wrapper` and `target_wrapper`. With wrappers around, a target is reached by an initiator through multiple socket bindings. As described in Section 3.2, a SCP is a list of sockets from an initiator to a target. In this example, `init` reaches `target` through two initiator socktets: `out1` and `out2`. Thus the SCP from `init` to `target` is [out1→out2]. With the correct SCP information, variable accesses for entangled variables are analyzed the same way as for direct communication.

7.2 Interconnected Communication

Interconnections such as routers and buses are common in SystemC models. In the example in Figure 7(b), `interconnect` forwards the communication from `init` to `target`. With an interconnecting module, a target is reached by an initiator through multiple initiator sockets, and thus the SCP contains multiple sockets. In Figure 7(b), the SCP from `init` to `target` is [out1→out2]. With the correct SCP information, variable accesses for entangled variables are analyzed the same way as for direct communication.

It is common that an interconnect module connects to multiple initiators and targets. An example is shown in Figure 8. Initiators `init1` and `init2` connect to targets `target1` and `target2` via `interconnect`. Because the compiler cannot statically identify which target a TLM-2.0 API call in the initiator `init1` is routed to during run time, our approach has to analyze both

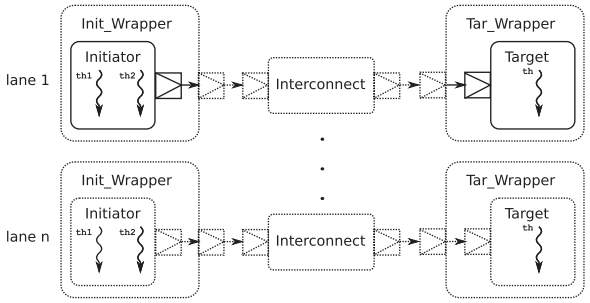


Fig. 9. Block diagram for the demonstration example.

possibilities. Specifically, `init1` reaches `target1` via SCP [out1→out2] and reaches `target2` via SCP [out1→out4], `init2` reaches `target1` via SCP [out3→out2] and reaches `target4` via SCP [out3→out4]. Variable accesses for entangled variables are analyzed the same way as for direct communication based on such SCP information.

8 EXPERIMENTS AND RESULTS

We have implemented the proposed static analysis approach as an extension of the SystemC compiler download from [4]. The project is written in C++ and uses the ROSE infrastructure [17] to build an AST of the input SystemC model. Based on the AST and the RISC internal representations, we are able to analyze sockets mappings, TLM-2.0 APIs and variable entanglements. We have evaluated our approach with demonstration and real-world examples. All the examples are SystemC TLM-2.0 approximately- or loosly-timed models with multiple threads and some parallelism potential. For evaluation, we measure the number of conflicts and the execution times under the sequential Accellera simulator (Seq) and the parallel RISC simulator (Par). Note that with the previous RISC compiler and simulator, none of these examples would actually compile. The compiler would output an error message about the use of TLM-2.0 constructs, such as unknown sockets and interface methods. Even if the model would pass the compiler, all pointers would result in conflicts with all other segments, leaving no threads available for parallel execution in the simulator. Thus, the examples would run only sequentially, similar as in the Accellera simulator. Our experiments execute on an Intel Xeon E3-1240 multi-core processor with 4 cores, 2-way hyperthreaded. The CPU frequency-scaling was turned off so as to provide accurate and repeatable results.

8.1 Demonstration Examples

The examples in this experiment are derived from the loosely-timed `LT` models in the Accellera SystemC library [18]. 36 examples are used with the following configurable variations:

(1) TLM-2.0 interface type: BTI, DMI or NBTI.
(2) communication type: Direct, Hierarchical (Hier) or Interconnected (Inter).
(3) varying number of lanes.

A generic block diagram is shown in Figure 9. The dotted lines indicate varied modules or socket connections. Despite the configurable variations, the number of threads in each initiator and target remain fixed. An initiator has two SC_THREADs where `th1` performs pure computation and `th2` performs communication, and a registered callback method for the backward NBTI path. A target has one SC_THREAD `th` and three registered callback methods for BTI, DMI and the forward NBTI path. The interconnect module has registered callbacks that route the communication. During communication, `Initiator::th2` accesses memory locations in both the initiator and the

Table 1. Results of BTI Examples from Accellera : Run-time(secs) and Speedup(%)

	Direct			Hierarchical			Interconnect		
Lanes	Seq	Par		Seq	Par		Seq	Par	
1	41.0	21.5	191%	41.2	21.5	192%	41.3	21.7	190%
2	80.8	22.5	359%	81.1	21.4	379%	81.7	22.5	363%
4	160.6	29.6	543%	162.0	29.9	542%	161.3	30.4	531%
8	320.8	59.2	542%	320.7	57.3	560%	320.7	58.0	553%

Table 2. Results of DMI Examples from Accellera: Run-time(secs) and Speedup(%)

	Direct			Hierarchical			Interconnect		
Lanes	Seq	Par		Seq	Par		Seq	Par	
1	41.7	21.6	193%	41.3	21.8	190%	41.4	21.7	191%
2	81.3	23.0	353%	81.1	22.4	362%	81.3	22.7	358%
4	161.2	30.8	523%	161.1	29.8	541%	161.2	29.5	546%
8	320.0	57.6	556%	322.0	58.4	551%	319.5	57.3	558%

Table 3. Results of NBTI Examples from Accellera: Run-time(secs) and Speedup(%)

	Direct			Hierarchical			Interconnect		
Lanes	Seq	Par		Seq	Par		Seq	Par	
1	40.2	20.1	200%	40.6	20.3	200%	40.5	20.3	200%
2	81.1	21.2	383%	80.7	20.8	388%	80.6	21.5	375%
4	159.2	28.7	554%	159.1	28.3	561%	161.2	28.9	558%
8	320.0	56.2	569%	320.1	57.2	559%	317.7	56.0	567%

Table 4. Percentage of Reduced False Variable Entanglements in the Demonstration Examples

	BTI			DMI			NBTI		
Lanes	Direct	Hier	Inter	Direct	Hier	Inter	Direct	Hier	Inter
1	33%	33%	33%	33%	33%	33%	33%	33%	33%
2	73%	73%	73%	73%	73%	73%	71%	71%	71%
4	88%	88%	88%	88%	88%	88%	87%	87%	87%
8	94%	94%	94%	94%	94%	94%	94%	94%	94%

target and thus has data conflicts with `Initiator::th1` and `Target::th`. The initiators, targets, wrappers and interconnects are respectively different instances of the same module types.

The sequential (Seq) and out-of-order parallel (Par) simulator run-times and speedups under different model configurations are shown in Table 1, 2 and 3. Table 4 shows the percentage of reduced false variable entanglements over all entanglements. Take the model with configuration *BTI+Inter+2-Lanes* as an example. An `Initiator` uses one pointer (in local module scope), `Interconnect` uses no pointer, and `Target` uses two pointers (one in local scope and the other in module scope). Since there are two lanes, the total number of pointers is six, and therefore there are 15 pairs of pointers in combination. Without the analysis for variable entanglement, all 15 pairs are considered as total conflicts. Now with our approach, the compiler is able to use context information and identify that only variables that belong to the same lane are entangled. Specifically, only four pairs of pointers are truly entangled. Thus 73% of entangled variable pairs are

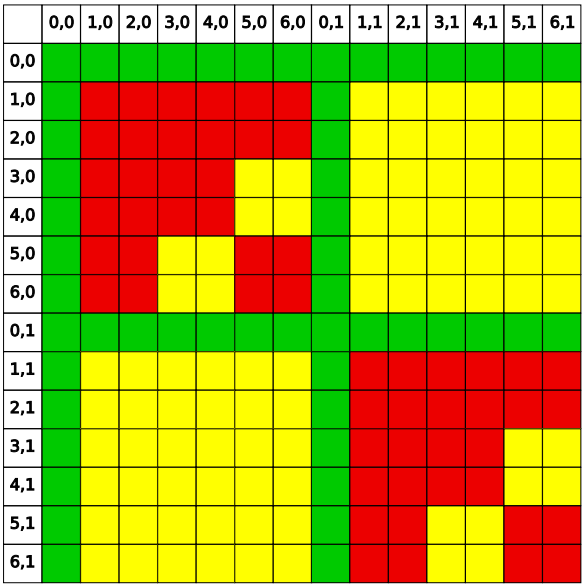


Fig. 10. Data conflict table for BTI+Inter+2-Lanes.

false entanglements and reduced by our approach. This results in much fewer conflicts in the data conflict table. Note that the numbers of pointers and truly entangled pointer pairs do not change with communication types. Therefore, the percentages of reduced false variable entanglements are the same for different communication types. On the other hand, the numbers of pointers and truly entangled pointer pairs are affected by interface types. Due to the backward path, the NBTI models have one more pointer in the callback method in `Initiator` than the BTI or DMI models. Therefore, the percentages of reduced false variable entanglements are the same for BTI and DMI, but slightly different for NBTI.

The data conflict table is shown in Figure 10. The indexes of the table have the form `(Segment ID, module Instance ID)`. For instance, `(2,0)` represents a segment with ID 2 and belongs to a thread of a module instance in the first lane, whereas `(2,1)` is the same segment with ID 2 but belongs to a thread of a module instance in the second lane. A red entry represents data conflicts, a green entry represents conflict free and a yellow entry represents the eliminated false conflicts by applying our SCP technique and variable entanglement analysis. The data conflict table shows that with our proposed static analysis, the number of data conflicts is reduced from 144 to 56. More than 60% of data conflicts are pointer-related false conflicts and are eliminated. The experimental results allow the following observations:

(1) Variable entanglement analysis largely reduces false conflicts due to pointers. Without the variable entanglement analysis, the pointers impose data conflicts with other segments and thus cause overwhelming false conflicts, as indicated by the yellow entries in Figure 10. Our approach is able to identify the exact memory location a pointer points to through the TLM-2.0 interfaces and thus enables precise data conflict analysis.
(2) The SCP technique allows instance-aware conflict analysis. Our SCP technique provides context information to the compiler to distinguish threads that belong to different module instances. Without the SCP technique, thread `th1` in the first `Initiator` module instance cannot run in parallel with `th2` in the second `Initiator` module instance due to false

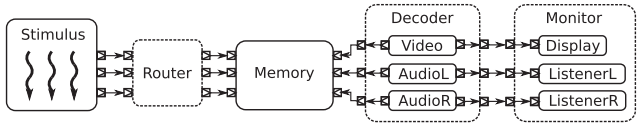


Fig. 11. Block diagram for the DVD player example.

Table 5. Results of DVD Player: Run-time(secs) and Speedup(%)

Interface	Direct			Hierarchical			Interconnect		
	Seq	Par		Seq	Par		Seq	Par	
BTI	208.1	73.8	282%	208.1	75.7	274%	208.4	74.8	278%
DMI	208.2	73.7	282%	208.5	75.5	276%	208.4	7.47	279%
NBTI	209.3	74.9	279%	209.4	75.6	277%	209.5	7.57	277%

conflicts. In the data conflict table, this for instance results in a false conflict between (1,0) and (3,1).

Given the novel analysis results, the OoO PDES simulator is able to execute all computation threads in the `Initiator` and `Target` in parallel and achieve a speedup of about 360% compared with the sequential execution. The ideal speedup of 400% is not achievable because of two main reasons: (a) the communication thread `Initiator::th2` cannot run in parallel with other threads due to data conflicts and causes a sequential bottleneck. According to Amdahl's law, this reduces the overall parallelism of the model (b) the OoO PDES simulator has a run-time overhead of dynamic checking. Other examples with different model configurations also demonstrate impressive speedups. A maximum speedup of over 550% is achieved for the 8-lane models. Overall, the demonstration examples show the correctness and effectiveness of the proposed static analysis for SystemC TLM-2.0 models.

8.2 DVD Player

Now we evaluate our approach using a real world DVD Player example which is similar to the one in Figure 1. The block diagram of the model is shown in Figure 11. `Stimulus` has three parallel threads that feed data into a memory `Memory` for the decoders to fetch. After decoding, the decoded results are passed to monitors to verify the correctness. One main difference from the model in Figure 1 is that there is only one memory module. This does not affect the parallelism of the model because different data flow lanes have their own memory locations inside `Memory`. Similar to the demonstration examples in Section 8.1, the DVD Player model is also configurable with the combination of following options: BTI/DMI/NBTI and Direct/Hierarchical/Interconnect. Note that the varied modules are represented by dashed lines. The two wrapper modules `Decoder` and `Monitor` are enabled under the "Hierarchical" configuration. `Router` is enabled under the "Interconnect" configuration. The results of run-time and speedup comparing to sequential simulation are shown in Table 5.

The results demonstrate that the model gets a speedup of around 280% under OoO PDES and is consistent over all model configurations. The consistency is reasonable because the TLM-2.0 communication and connection do not have impact on the threads' behavior and thus will not affect speed much. It is also notable that the theoretical maximum speedup of 300% is not achieved for this three-lane model. This is explainable because the OoO PDES scheduler needs to perform the scheduling and to decide thread dispatching order and thus incurs overhead. Nevertheless, 280% is an impressive value for a three-lane model. Note that this is also better than the 242% speedup reported in [16] for a comparable TLM-1.0 DVD player model.

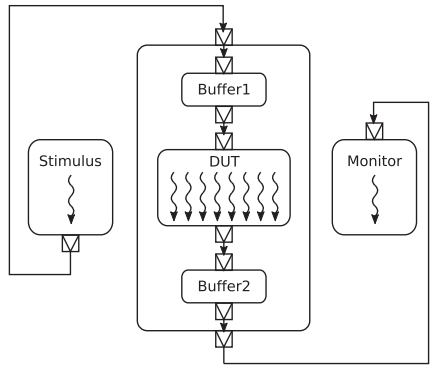


Fig. 12. Block diagram for the MandelBrot renderer example.

Table 6. Results of Mandelbrot Renderer: Run-time(secs) and Speedup(%)

Interface	Seq	Par	Speedup
BTI	77.59	18.66	416%
DMI	77.64	18.45	421%
NBTI	77.60	18.50	419%

The results confirm the correctness and effectiveness of the proposed static analysis for SystemC TLM-2.0 models.

8.3 Mandelbrot Renderer

The Mandelbrot renderer is a parallel image rendering application to compute the Mandelbrot set. The platform architecture is shown in Figure 12. The DUT module hosts eight parallel renderer threads. Each renderer thread computes a Mandelbrot image in a given area and sends the result to the controller thread of DUT. The controller merges all the results and sends it out, then starts to wait for new frames. Again, three experiments are performed on the the Mandelbrot Renderer example with different communication types: BTI, DMI and NBTI. The results of run-time and speedup comparing to sequential simulation are shown in Table 6.

As shown in the results, the speedup between OoO PDES and sequential simulation under both communication types are around 420%. The naive maximum speedup of 800% is not achieved because there are only 4 floating point units (FPU) in the processor and thus the eight computationally intensive renderer threads are not able to run totally in parallel[3]. Considering the restriction of hyperthreading, the 420% speedup is impressive on a 4 core machine. The results of this experiment demonstrate again that the proposed static analysis is effective and correct to support OoO PDES of SystemC TLM-2.0 models using BTI, DMI and NBTI.

8.4 Bitcoin Miner

Our third real-world example is a SystemC TLM-2.0 model of a Bitcoin miner. Bitcoin miners create Bitcoins by solving math problems using a computation-intensive cryptographic hashing

[3]In the BTI example, the user time of sequential simulation is 77.59 seconds which is smaller than 92.04 seconds for parallel simulation. This shows that each hyperthread during parallel simulation is running longer due to the contention in the FPU.

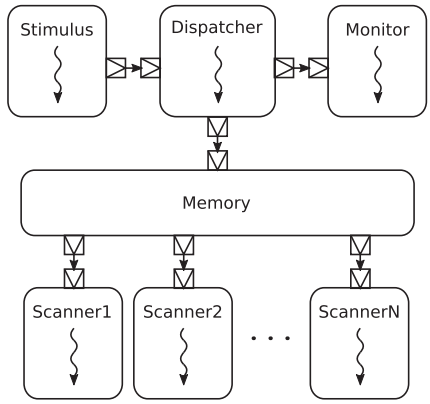


Fig. 13. Block diagram for the Bitcoin miner example.

Table 7. Results of Bitcoin Miner: Run-time(secs) and Speedup(%)

	BTI			DMI			NBTI		
Scanners	Seq	Par		Seq	Par		Seq	Par	
1	1903.02	1902.17	100%	1902.64	1908.29	100%	1889.93	1899.13	99%
2	1680.54	858.06	196%	1675.68	855.61	196%	1677.73	850.95	197%
4	1885.32	508.24	371%	1890.30	507.47	372%	1895.60	506.56	373%
8	1944.04	424.02	458%	1948.76	420.77	464%	1934.75	423.59	457%
16	2282.64	506.33	450%	2274.40	507.76	448%	2283.62	498.07	458%
64	3169.93	696.75	455%	3161.10	698.49	452%	3169.84	688.85	460%
256	6827.77	2315.08	295%	6824.44	2056.17	332%	6818.38	2236.09	305%

algorithm. Figure 13 shows the structure of the Bitcoin miner model. `Stimulus` prepares the math problems and sends them to `Dispatcher`. `Dispatcher` divides the received problems into multiple sub-problems and stores them into `Memory`. From there, a `Scanner` picks a corresponding sub-problem and tries to solve it using a hashing algorithm. Once done, the `Scanner` stores the result into `Memory` and waits for a new sub-problem. `Dispatcher` fetches the result from `Memory` and sends it to `Monitor`. Note that in the Bitcoin miner model all `Scanners` are independent and their threads are able to run in parallel with our approach.

We perform experiments with different number of `Scanners`: 1, 2, 4, 8, 16, 64, 256 and different communication types: BTI, DMI, NBTI. The results are shown in Table 7. Similar to previous experiments, the communication type does not significantly affect the simulation speeds. Because of the high parallelism level of Bitcoin miner, the speedup between OoO PDES and sequential simulation keeps increasing and reaches a maximum of around 460% when there are 8 `Scanners`. Although the host processor has eight hyperthreads, a naive speedup of 800% cannot be achieved due to the contention in the FPU. The speedup remains at the same level with more `Scanners` and even drops when there are 256 `Scanners`. This is because of the dramatically increased context switching between threads. The context switching is also a major factor that significantly affects the sequential and parallel simulation time when there are more than 8 scanners. Considering all the hardware restrictions, this experiment still demonstrates the effectiveness and correctness of our proposed static analysis for supporting OoOPDES of SystemC TLM-2.0 models using BTI, DMI and NBTI.

9 CONCLUSION

In this paper, we propose a compiler-based approach to statically analyze SystemC TLM-2.0 loosely-timed (LT) and approximately-timed (AT) models. The analysis is essential to simulate the model under OoO PDES. In the proposed approach, an accurate SG is first built with SCP information for TLM-2.0 interface function calls. Then, the compiler analyzes the variable accesses for entangled variables, precisely identifying potential conflicts. Our experiments demonstrate the correctness and effectiveness of the approach with demonstration examples from Accellera and three real world examples: DVD Player, Mandelbrot Renderer and Bitcoin Miner.

In future work, we plan to apply also a hybrid approach to analyze dynamic socket bindings.

ACKNOWLEDGMENT

This work has been supported in part by substantial funding from Intel Corporation for the project titled "Scaling the Recoding Infrastructure for Parallel SystemC Simulation". The authors thank Intel Corporation for the valuable support.

REFERENCES

[1] IEEE Standard 1666-2011 for Standard SystemC® Language Reference Manual, IEEE Computer Society, January 2012.
[2] W. Chen, X. Han, C. W. Chang, G. Liu, and R. Dömer. 2014. Out-of-order parallel discrete event simulation for transaction level models. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 33, 12 (2014), 1859–1872.
[3] R. Dömer. 2016. Seven obstacles in the way of standard-compliant parallel systemc simulation. *IEEE Embedded Systems Letters* 8, 4 (December 2016), 81–84.
[4] Lab for Embedded Computer Systems (LECS), Recoding Infrastructure for SystemC [Online]. Available: http://www.cecs.uci.edu/~doemer/risc.html#RISC050.
[5] R. Fujimoto. 1990. Parallel discrete event simulation. *Commun. ACM* 33 (Oct. 1990), 3053.
[6] T. Schmidt, Z. Cheng, and R. Dömer. 2018. Port call path sensitive conflict analysis for instance-aware parallel systemc simulation. *Proceedings of Design, Automation and Test in Europe*, Dresden, Germany, March 2018.
[7] T. Schmidt, G. Liu, and R. Dömer. 2017. Exploiting thread and data level parallelism for ultimate parallel systemc simulation. *Proceedings of the Design Automation Conference 2017*, Austin, TX, June 2017.
[8] A. Kaushik. Patel HD (2013) SystemC-clang: An open-source framework for analyzing mixed-abstraction SystemC models. *Proceedings of the forum on specification and design languages (FDL)*, Paris.
[9] J. Viitanen, P. Sjvöall, M. Viitanen, T. D. Hämäläinen, and J. Vanne. 2016. Distributed systemc simulation on manycore servers. In *IEEE Nordic Circuits and Systems Conference (NORCAS)*, (2016), 1–6.
[10] J. H. Weinstock, R. Leupers, G. Ascheid, D. Petras, and A. Hoffmann. 2016. SystemC-link: Parallel SystemC simulation using time-decoupled segments. In *Proceedings of Design, Automation and Test in Europe*, 2016.
[11] D. Becker, M. Moy, and J. Cornet. 2016. Parallel simulation of loosely timed SystemC/TLM programs: Challenges raised by an industrial case study. In *Electronics* 5, 2 (2016), 22.
[12] M. Moy. 2013. Parallel programming with SystemC for loosely timed models: A non-intrusive approach. In *Proceedings of Design, Automation and Test in Europe*, 2013.
[13] P. Combes, E. Caron, F. Desprez, B. Chopard, and J. Zory. 2008. Relaxing synchronization in a parallel SystemC kernel. In *IEEE International Symposium on Parallel and Distributed Processing with Applications*, 2008.
[14] DOULOS tutorial: Getting Started with TLM-2.0 [Online]. Available: https://www.doulos.com/knowhow/systemc/tlm2/tutorial__1/.
[15] OSCI TLM-2.0 LANGUAGE REFERENCE MANUAL, the Open SystemC Initiative (OSCI), July 2009.
[16] R. Dömer, G. Liu, and T. Schmidt. 2017. Parallel simulation. chapter 17 in "Handbook of Hardware/Software Codesign" by S. Ha and J. Teich, Springer Netherlands, Dordrecht, June 2017. (ISBN 978-94-017-7358-4).
[17] ROSE User Manual: A Tool for Building Source-to-Source Translators Draft User Manual (version 0.9.10.231), Daniel Quinlan, Chunhua Liao, Thomas Panas, Robb Matzke, Markus Schordan, Rich Vuduc, and Qing Yi, Lawrence Livermore National Laboratory, April 19, 2019.
[18] Accellera SystemC supplemental material [Online]. Available: https://www.accellera.org/downloads/standards/systemc.

Received April 2019; revised May 2019; accepted July 2019