On the Limits of Standard-compliant Parallel Simulation of the IEEE SystemC Language

Forum on specification & Design Languages
Keynote
Rainer Dömer
doemer@uci.edu
Center for Embedded and Cyber-Physical Systems
University of California, Irvine

IEEE Standard 1666-2011
• The SystemC Language
  – official standard
  – de-facto standard
• for
  – modeling
  – simulation
• of systems containing
  – hardware
  – software
➢ Keynote Focus
  ➢ Parallelism in models
  ➢ Parallelism in simulation
  ➢ Standard compliance

(c) 2018 R. Doemer, CECS
Discrete Event Simulation (DES)

- SystemC uses DES
  - Concurrent threads of execution
  - Managed by a central scheduler
  - Driven by events and time advances
    - Delta cycle
    - Time cycle
  - Partial temporal order with barriers
- Accellera Reference Simulator
  - Proof-of-concept implementation of IEEE 1666-2011 standard
  - A single thread is active at any time
    - Does not exploit parallelism
    - Cannot utilize multiple cores
  - Sequential simulation is slow

Approaches for Faster Simulation

Improved Modeling Techniques
- Transaction-level modeling (TLM)
- TLM temporal decoupling
- Savoiu et al. [MEMOCODE'05]
- Razaghi et al. [ASPDAC'12]

Sequential DE simulation is slow

Hardware-based Acceleration
- Sirowy et al. [DAC'10]
- Nanjundappa et al. [ASPDAC'10]
- Sinha et al. [ASPDAC'12]

Distributed Simulation
- Chandy et al. [TSE'79]
- Huang et al. [SIES'08]
- Chen et al. [CECS'11]

SMP Parallel Simulation
- Fujimoto [CACM'90]
- Chopard et al. [ICCS'06]
- Ezudheen et al. [PADS'09]
- Mello et al. [DATE'10]
- Schumacher et al. [CODES'11]
- Chen et al. [TCAD'14]
- Yun et al. [TCAD'12]
- Schmidt et al. [DAC'17]
- and many others
Parallel Discrete Event Simulation (PDES)

- Parallel DES [Fujimoto1990]
  - Threads execute in parallel \textit{iff}
    - in the same delta cycle, \textit{and}
    - in the same time cycle
  \quad Order of magnitude speed up!
  \quad Problem solved!?
  \quad Not quite!
  \quad What about host platforms?
    - Multi- and many-core hosts are readily available
  \quad What about accuracy?
    - Is achievable with careful analysis
  \quad What about standard compliance?
    - That’s where the problem is!

Problem Definition

- Given
  - Embedded systems are parallel
  - SystemC is suitable and popular for system design
  - Models exhibit explicit thread-level parallelism
  - Multi- and many-core host platforms are readily available
- Design
  - Fast Parallel Discrete Event Simulation
  - For the SystemC language
- Optimize
  - Maximize compliance with the IEEE 1666-2011 standard
  \quad Why is this difficult?
    - 7 Obstacles stand in the way of standard-compliant parallel SystemC simulation [ESL'16]
### Obstacle 1: Co-Routine Semantics

- **Fact:** IEEE 1666-2011 requires *co-operative multitasking*
  - Quotes from Section “4.2.1.2 Evaluation phase” (pages 17, 18):
    - Since process instances execute without interruption, *only a single process instance can be running at any one time*. [...] A process shall not pre-empt or interrupt the execution of another process. This is known as *co-routine semantics* or *co-operative multitasking*.
    - [...] The scheduler is *not pre-emptive*. An application can assume that a method process will execute in its entirety without interruption, and a thread or clocked thread process will execute the code between two consecutive calls to function `wait` without interruption.

- **Problem:** Uninterrupted execution guarantee
  - An implementation running on a machine that provides hardware support for concurrent processes may permit two or more processes to run concurrently, provided that the behavior appears identical to the co-routine semantics defined in this subclause. In other words, the implementation would be obliged to analyze any dependencies between processes and to constrain their execution to match the co-routine semantics.

### Parallel Discrete Event Simulation (PDES)

- **Parallel DES [Fujimoto 1990]**
  - Threads execute in parallel *iff*
    - in the same delta cycle, and
    - in the same time cycle
  - Order of magnitude speed up!
  - **IEEE 1666 Requirement:**
    - “The scheduler is *not pre-emptive*.”

```c
int x; // shared variable

void thread1() { x = 0; x = x + 1; cout << x; }
void thread2() { x = 7; x = x * 6; cout << x; }
```

- **SystemC:** guaranteed safe!
- **PDES:** not safe! (race condition)
Obstacle 1: Co-Routine Semantics

- Fact: IEEE 1666-2011 requires *co-operative multitasking*
  - Quotes from Section "4.2.1.2 Evaluation phase" (pages 17, 18):
    - Since process instances execute without interruption, only a single process instance can be running at any one time. [...] A process shall not pre-empt or interrupt the execution of another process. This is known as *co-routine semantics* or *co-operative multitasking*.
    - The scheduler is not pre-emptive. An application can assume that a method process will execute in its entirety without interruption, and a thread or clocked thread process will execute the code between two consecutive calls to function wait without interruption.

- Problem: Uninterrupted execution guarantee
  - Processes at the same time (T, Δ) may execute in parallel
    - Model designer must write thread safe code, avoid race conditions
    - Parallel systems, parallel models, parallel programming

- Proposal: Explicitly allow parallel execution, preemption
  - Processes at the same time (T, Δ) may execute in parallel
    - Model designer must write thread safe code, avoid race conditions
    - Parallel systems, parallel models, parallel programming

Obstacle 2: Simulator State

- Fact: Discrete Event Simulation (DES) is presumed
  - Example from IEEE 1666-2011, page 31: `sysc/kernel/sc_simcontext.h`
    ```
    [...]
    bool sc_pending_activity_at_current_time();
    bool sc_pending_activity_at_future_time();
    bool sc_pending_activity();
    bool sc_time_to_pending_activity();
    [...]
    ```

- Problem: Parallel Discrete Event Simulation (PDES) is different from sequential DES
  - After elaboration, there may be multiple running threads
  - Scheduling may happen while some threads are still running

- Proposal: Carefully review simulator state primitives and revise as needed for PDES
  - Adapt the functions and APIs for parallel execution semantics
  - Entire accessible simulator state needs attention...
Obstacle 2: Simulator State

- **Fact:** Discrete Event Simulation (DES) is presumed
- **Problem:** Parallel Discrete Event Simulation (PDES) is different from sequential DES
- **Proposal:** Carefully review simulator state primitives and revise as needed for PDES
  - *Entire* accessible simulator state needs attention
  - Special consideration for very strict semantics, e.g. debugging:
    
    Quote from IEEE 1666-2011, Section “4.2.1.2 Evaluation phase” (page 17):
    > The order in which process instances are selected from the set of runnable processes is implementation defined. However, if a specific version of a specific implementation runs a specific application using a specific input data set, the order of process execution shall not vary from run to run.
  - Sequential DES can remain valid as a special case of PDES
    - While PDES typically runs up to \( n \) threads in parallel, where \( n \) = number of cores on the host, we can set \( n = 1 \) to mimic the classic DES case

Obstacle 3: Lack of Thread Safety

- **Fact:** Primitives are generally not multi-thread safe
  - Suspicious example from IEEE 1666-2011, page 194:
    ```
    [...] 
    sc_length_param length10(10); 
    sc_length_context cntxt10(length10); // length10 now in context 
    sc_int_base int_array[2]; // Array of 10-bit integers 
    [...] 
    ```
  - **Problem:** Parallel execution may lead to race conditions
    - Race conditions result in non-deterministic/undefined behavior
    - Explicit protection (e.g. by mutex locks) is cumbersome
    - Identifying problematic constructs is difficult
      - Example: *class sc_context*, commented as “co-routine safe”
  - **Proposal:** Require *all* primitives to be multi-thread safe
    - Carefully revise the proof-of-concept SystemC library
    - Encouraging item: *async_request_update* is thread-safe!
      - See “5.15 sc_prim_channel”, IEEE 1666-2011, page 121
Obstacle 4: Class sc_channel

- Proposal: Class `sc_channel`, derived from `sc_module`
  - Module encapsulates computation (hosts threads/processes)
  - Channel encapsulates communication (implemented interfaces)
  - Q: Why do we need channels? A: Thread safe communication!
    - Example: Blocking write in primitive channel `sc_fifo.h`

```cpp
template <class T> inline
void sc_fifo<T>::write( const T& val_ )
{
  sc_stacked_lock l(m_mutex); // lock the channel mutex
  while( num_free() == 0 ) {
    sc_core::wait( m_data_read_event );
    m_num_written ++;
    buf_write( val_ );
    request_update();
  }
}
```

- Race condition between `num_free` and `m_num_written`
- Prevented by locking `m_mutex` of this channel instance
- Channel acts as a `monitor` for multi-thread safe communication
Obstacle 5: TLM-2.0

- Fact: Channel concept has disappeared

- Problem:
  Where is the channel?
  - Interface methods are well-defined, but not contained
  - Separation of concerns “Computation ≠ Communication” principle is broken
On the Limits of Standard-compliant Parallel Simulation of the IEEE SystemC Language

FDL'18 Keynote, Munich, Germany

Obstacle 5: TLM-2.0

- Fact: Channel concept has disappeared

- Problem: Where is the channel?
  - Interface methods are well-defined, but not contained
  - Separation of concerns "Computation ≠ Communication" principle is broken
  - Naïve Proposal: Encapsulate communication methods in channels

Obstacle 6: Sequential Mindset

- Fact: SC_METHOD is preferred over SC_THREAD, context switches are considered overhead
  - IEEE 1666-2011, Section 5.2.11 on threads (page 44):

- Problem: Sequential modeling is encouraged
  - However, systems are parallel by nature, so should be models
  - Avoiding context switches is the wrong optimization criterion

- Proposal: Use actual threads, avoid SC_METHOD, identify dependencies among threads
  - Promote parallel mindset, with true thread-level parallelism
    - Speed due to parallel execution, not due to fewer context switches
  - Explicitly express task relations (use e.notify(), wait(e))
    - Synchronize, communicate through events and channels
Obstacle 7: Temporal Decoupling

- Fact: TD is designed to speed up sequential DES
  - IEEE 1666-2011, Section 12.1 on "TLM-2.0 global quantum" (page 453):
    - Abstraction trades off accuracy for higher simulation speed
- Problem: PDES is a different foundation than DES
  - TD design assumptions are not necessarily true for PDES
  - Global time quantum is a technical obstacle (race condition)
- Proposal: Reevaluate costs/benefits, redesign if needed
  - Analyze TD idea for PDES, adopt advantages, drop drawbacks
    - Avoid tlm_global_quantum, promote wait(time)
  - Consider the use of a compiler to optimize scheduling, timing
    - Out-of-Order PDES [TCAD’14] is one solution...

Temporal decoupling permits SystemC processes to run ahead of simulation time for an amount of time known as the time quantum and is associated with the loosely-timed coding style. Temporal decoupling permits a significant simulation speed improvement by reducing the number of context switches and events.

Now what?

- Seven Obstacles stand in the Way of Standard-Compliant Parallel SystemC Simulation
  - Truly parallel and truly compliant SystemC appears elusive given the current IEEE standard
SystemC Evolution?

- Seven Obstacles stand in the Way of Standard-Compliant Parallel SystemC Simulation
  - SystemC Evolution Day 2016 [IEEE ESL'16]
  - Let’s overcome the identified 7 obstacles!
    - Move up from DES to PDES
    - Adopt a parallel mindset, expose and exploit parallelism
    - Apply the principle of separation of concerns
      - Modules encapsulate computation
      - Channels encapsulate communication
    - Simulate models faster with parallel execution semantics
  - SystemC must evolve in a major revision (3.x)
    - C++11 already has built-in support for multithreading
    - SystemC must embrace true parallelism

Maximum Compliance with Standard

- Seven Obstacles stand in the Way of Standard-Compliant Parallel SystemC Simulation
  - SystemC Evolution Day 2016 [IEEE ESL’16]
- In absence of major changes to SystemC standard, let’s make the best of it
  - Accept SystemC as it is (well, most of it)
  - Build the best parallel SystemC simulator possible
  - Aim for maximum compliance with the standard
- We took this risk, and created RISC!
Recoding Infrastructure for SystemC (RISC)

- Advanced Parallel SystemC Simulation
  - Aggressive PDES on many-core host platforms
  - Maximum compliance with IEEE SystemC semantics
- Introduction of a Dedicated SystemC Compiler
  - Advanced conflict analysis for safe parallel execution
  - Automatic model instrumentation and code generation
- Parallel SystemC Simulator
  - Out-of-order parallel scheduler, multi-thread safe primitives
  - Multi- and many-core host platforms (e.g. Intel® Xeon Phi™)
- Open Source
  - Freely available for evaluation and collaboration
  - Thanks to Intel Corporation!

Recoding Infrastructure for SystemC (RISC)

- **Out-of-Order Parallel DES**
  - Threads execute in parallel *iff*
    - in the same delta cycle, and
    - in the same time cycle,
    - *OR if there are no conflicts!*
  - Breaks synchronization barrier!
  - Threads run as soon as possible, even ahead of time.
  - Significantly higher speedup!
    - Results at [DATE’12], [IEEE TCAD’14]
  - RISC compiler fully preserves…
    - Cause and effect relationship
    - Accuracy in results and timing
Recoding Infrastructure for SystemC (RISC)

- **Out-of-Order PDES Key Ideas**
  1. Dedicated *SystemC compiler* with advanced model analysis
     - Static conflict analysis based on Segment Graphs
  2. *Parallel simulator* with out-of-order scheduling
     - Fast decision making at run-time, optimized mapping

- **Fundamental Data Structure: Segment Graph**
  - Key to semantics-compliant out-of-order execution [DATE’12]
  - Key to prediction of future thread state [DATE’13]
    - “Optimized Out-of-Order Parallel DE Simulation Using Predictions”
  - Key to May-Happen-in-Parallel Analysis [DATE’14]
    - “May-Happen-in-Parallel Analysis based on Segment Graphs for Safe ESL Models” (*Best Paper Award*)
  - Combined: “OoO PDES for TLM” [IEEE TCAD’14]
    - Comprehensive summary with HybridThreads extension

Dedicated SystemC Compiler

- **RISC Software Stack**
  - *Recoding Infrastructure for SystemC*
    - C/C++ foundation
    - ROSE compiler (from LLNL)
  - ROSE-based tools

- **ROSE Internal Representation**
  - Explicit support for
    - Source code analysis
    - Source-to-source transformations

Source:
Lawrence Livermore National Laboratory (LLNL)

(c) 2018 R. Doemer, CECS
On the Limits of Standard-compliant Parallel Simulation of the IEEE SystemC Language

FDL’18 Keynote, Munich, Germany

Dedicated SystemC Compiler

- RISC Software Stack
  - Recoding Infrastructure for SystemC
    - SystemC Internal Representation
  - Class hierarchy to represent SystemC objects

1) Segment Graph
2) Parallel access conflict analysis

Step 1: Build a Segment Graph
Dedicated SystemC Compiler

- **Segment Graph**
  - *Segment Graph* is a directed graph
    - Nodes: *Segments*
      - Code statements executed between two scheduling steps
        - Expression statements
        - Control flow statements (*if*, *while*, …)
        - Function calls
    - Edges: *Segment boundaries*
      - Primitives that trigger scheduler entry
        - `wait(event)`
        - `wait(time)`
  - Segment Graph is built automatically by the compiler [TCAD’14]
    - From the model source code
    - Via Abstract Syntax Tree and Control Flow Graph

Dedicated SystemC Compiler

- **RISC Software Stack**
  - *Reencoding Infrastructure for SystemC*
    1) Segment Graph construction
    2) Parallel access conflict analysis
    3) Model instrumentation

(c) 2018 R. Doemer, CECS
Dedicated SystemC Compiler

- Segment Conflict Analysis
  - Need to comply with SystemC LRM [IEEE Std 1666™]
  - Cooperative (or co-routine) multitasking semantics
    - "process instances execute without interruption"
    - System designer "can assume that a method process will execute in its entirety without interruption"
    - A parallel implementation "would be obliged to analyze any dependencies between processes and constrain their execution to match the co-routine semantics."
  - Must avoid race conditions when using shared variables!
    - Prevent conflicting segments to be scheduled in parallel

SystemC Compiler and Simulator

- Compiler and Simulator work hand in hand
  - Compiler performs conservative static analysis
  - Analysis results are passed to the simulator
  - Simulator can make safe scheduling decisions quickly

  ➢ Automatic Model Instrumentation
  ➢ Static analysis results are inserted into the source code
Parallel SystemC Simulator

- Simulator kernel with Out-of-Order Parallel Scheduler
  - Conceptual OoO PDES execution

Experiments and Results

- DVD Player Example
  - Parallel video and audio decoding with different frame rates

1: SC_MODULE(VideoCodec)
2: { sc_port<i_receiver> p1;
3:   sc_port<i_sender> p2;
4:   ...
5:   while(1) {
6:      p1->receive(&inFrm);
7:      outFrm = decode(inFrm);
8:      wait(33330, SC_US);
9:      p2->send(outFrm);
10: } }
Experiments and Results

• DVD Player Example
  – Parallel video and audio decoding with different frame rates
    1. Real time schedule: fully parallel
    
    2. Reference simulator schedule (DES)
        
    3. Synchronous parallel schedule (PDES)
Experiments and Results

- DVD Player Example
  - Parallel video and audio decoding with different frame rates
    1. Real time schedule: fully parallel
    4. Out-of-order parallel schedule (OoO PDES)

Experiments and Results

- DVD Player Example
  - Parallel video and audio decoding with different frame rates
- Simulator Run Times
  - 4-core Intel® Xeon® CPU at 3.4 GHz
  - RISC v0.2.1, Posix-threads

<table>
<thead>
<tr>
<th></th>
<th>DES</th>
<th>PDES</th>
<th>OoO PDES</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 sec stream</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Run Time</td>
<td>6.98 s</td>
<td>4.67 s</td>
<td>2.94 s</td>
</tr>
<tr>
<td>CPU Load</td>
<td>97%</td>
<td>145%</td>
<td>238%</td>
</tr>
<tr>
<td>Speedup</td>
<td>1 x</td>
<td>1.49 x</td>
<td>2.37 x</td>
</tr>
<tr>
<td>100 sec stream</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Run Time</td>
<td>68.21 s</td>
<td>45.91 s</td>
<td>28.13 s</td>
</tr>
<tr>
<td>CPU Load</td>
<td>100%</td>
<td>149%</td>
<td>251%</td>
</tr>
<tr>
<td>Speedup</td>
<td>1 x</td>
<td>1.49 x</td>
<td>2.42 x</td>
</tr>
</tbody>
</table>
Experiments and Results

- Mandelbrot Renderer (Graphics Pipeline Application)
  - Mandelbrot Set
    - Mathematical set of points in complex plane
      - Two-dimensional fractal shape
    - High computation load
      - Recursive/iterative function
    - Embarrassingly parallel
      - Parallelism at pixel level
  - SystemC Model
    - TLM abstraction
    - Horizontal image slices
    - Highly configurable
    - Parallelism parameter from 1 to 256 slices

Experiments and Results

- Mandelbrot Renderer (Graphics Pipeline Application)
  - Simulated Graphics Demonstration
    (when network delays prevent actual graphical demo)
Experiments and Results

- **Mandelbrot Renderer (Graphics Pipeline Application)**
  - Simulator run times on 16-core Intel® Xeon® multi-core host
  - 2 CPUs at 2.7 GHz, 8 cores each, 2-way hyper-threaded
  - RISC V0.2.1, Posix-threads

<table>
<thead>
<tr>
<th>Parallel Slices</th>
<th>DES Run Time</th>
<th>CPU Load</th>
<th>PDES Run Time</th>
<th>CPU Load</th>
<th>Speedup</th>
<th>OOO PDES Run Time</th>
<th>CPU Load</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>162.13 s</td>
<td>99%</td>
<td>162.06 s</td>
<td>100%</td>
<td>1.00 x</td>
<td>161.90 s</td>
<td>100%</td>
<td>1.00 x</td>
</tr>
<tr>
<td>2</td>
<td>162.19 s</td>
<td>99%</td>
<td>96.50 s</td>
<td>168%</td>
<td>1.68 x</td>
<td>96.48 s</td>
<td>168%</td>
<td>1.68 x</td>
</tr>
<tr>
<td>4</td>
<td>162.56 s</td>
<td>99%</td>
<td>54.00 s</td>
<td>305%</td>
<td>3.01 x</td>
<td>53.85 s</td>
<td>304%</td>
<td>3.02 x</td>
</tr>
<tr>
<td>8</td>
<td>163.10 s</td>
<td>99%</td>
<td>29.89 s</td>
<td>592%</td>
<td>5.46 x</td>
<td>30.05 s</td>
<td>589%</td>
<td>5.43 x</td>
</tr>
<tr>
<td>16</td>
<td>164.01 s</td>
<td>99%</td>
<td>19.03 s</td>
<td>1050%</td>
<td>10.06 x</td>
<td>20.08 s</td>
<td>997%</td>
<td>8.17 x</td>
</tr>
<tr>
<td>32</td>
<td>165.89 s</td>
<td>99%</td>
<td>11.78 s</td>
<td>2082%</td>
<td>14.08 x</td>
<td>11.99 s</td>
<td>2023%</td>
<td>13.84 x</td>
</tr>
<tr>
<td>64</td>
<td>170.32 s</td>
<td>99%</td>
<td>9.79 s</td>
<td>2607%</td>
<td>17.40 x</td>
<td>9.85 s</td>
<td>2608%</td>
<td>17.29 x</td>
</tr>
<tr>
<td>128</td>
<td>174.55 s</td>
<td>99%</td>
<td>9.34 s</td>
<td>2793%</td>
<td>18.69 x</td>
<td>9.39 s</td>
<td>2787%</td>
<td>18.59 x</td>
</tr>
<tr>
<td>256</td>
<td>185.47 s</td>
<td>100%</td>
<td>8.91 s</td>
<td>2958%</td>
<td>20.82 x</td>
<td>8.90 s</td>
<td>2964%</td>
<td>20.84 x</td>
</tr>
</tbody>
</table>

- **Many-Core Target Platform: Intel® Xeon Phi™**
  - Many Integrated Core (MIC) architecture
    - 1 Coprocessor 5110P CPU at 1.052 GHz
    - 60 physical cores with 4-way hyper-threading
      - Appears as regular Linux host with 240 cores
    - Up to 8 lanes available for vector processing
  - RISC extended for exploiting 2 types of parallelism
    - Out-of-Order PDES: thread-level parallelism
    - Intel® compiler SIMD: data-level parallelism
  - RISC SIMD Advisor identifies functions with data-level parallelism suitable for SIMD vectorization
  - DAC '17 paper:
    "Exploiting Thread and Data Level Parallelism for Ultimate Parallel SystemC Simulation"
Experiments and Results

- Many-Core Target Platform: Intel® Xeon Phi™
  - Exploiting thread- and data-level parallelism [DAC’17]
  - Mandelbrot renderer (graphics pipeline application)
- Experimental Results:

<table>
<thead>
<tr>
<th>PAR</th>
<th>MT</th>
<th>SIMD</th>
<th>MT+SIMD</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.00</td>
<td>6.92</td>
<td>6.94</td>
</tr>
<tr>
<td>2</td>
<td>1.68</td>
<td>6.92</td>
<td>11.77</td>
</tr>
<tr>
<td>4</td>
<td>3.04</td>
<td>6.92</td>
<td>21.19</td>
</tr>
<tr>
<td>8</td>
<td>5.84</td>
<td>6.92</td>
<td>40.10</td>
</tr>
<tr>
<td>16</td>
<td>11.37</td>
<td>6.92</td>
<td>72.52</td>
</tr>
<tr>
<td>32</td>
<td>21.32</td>
<td>6.91</td>
<td>137.21</td>
</tr>
<tr>
<td>64</td>
<td>41.07</td>
<td>6.90</td>
<td>208.41</td>
</tr>
<tr>
<td>128</td>
<td>46.29</td>
<td>6.89</td>
<td>212.96</td>
</tr>
<tr>
<td>256</td>
<td>49.90</td>
<td>6.87</td>
<td>194.19</td>
</tr>
</tbody>
</table>

Increasing degree of parallelism (PAR = number of threads) reaches a combined multi-threading (MT) and data-level (SIMD) speedup of up to 212x!

RISC Open Source Software

- RISC Compiler and Simulator are freely available
  - http://www.cecs.uci.edu/~doemer/risc.html#RISC042
    - Installation notes and script: INSTALL, Makefile
    - Open source tar ball: risc_v0.4.2.tar.gz
    - Docker script and container: Dockerfile
    - Doxygen documentation: RISC API, OOPSC API
    - Tool manual pages: risc, simd, visual,...
    - BSD license terms: LICENSE
  - Companion Technical Report

- Docker container:
  - https://hub.docker.com/r/ucirvinelecs/risc/
    - bash# docker pull ucirvinelecs/risc
    - bash# docker run -it ucirvinelecs/risc [dockeruser]
      - [dockeruser]# cd demodir
      - [dockeruser]# make test
Ongoing Efforts: Scaling RISC

- Support for Industrial Sizes and Design Flows
  1. New concept of Partial Segment Graphs (PSG)
     - File hierarchies with multiple translation units
     - Support for 3rd party libraries, IP protection
  2. Improved compiler analysis for less false conflicts
     - Port-Call-Path technique identifies instances [DATE’18]
     - Reference type analysis identifies target variables
  3. Evaluation of RISC in industry
     - “Big example”, very large SystemC model at RTL abstraction
     - Integration with Simics virtual platforms
  4. Support for TLM-2.0
     - Pro: Part of SystemC standard, needed for wide RISC adoption
     - Con: Obstacle 5!
       No channel, unprotected execution in foreign territory

Scaling RISC: Overcoming Obstacle 5

- SystemC TLM-2.0
  - Initiators and Targets
    - Sockets
    - Forward path
    - Backward path
    - Shared transaction object
    - DMI bypass
  - Well-defined Socket API
    1. _b_transport()
    2. _nb_transport_fw()
    3. _nb_transport_bw()
    4. _transport_dbg()
    5. get_direct_mem_ptr()
    6. invalidate_direct_mem_ptr()
Scaling RISC: Overcoming Obstacle 5

• Classic TLM: Producer-Consumer Example

- Threads operate in their own modules or protected channels
  - Well-behaved execution in safe execution contexts
  - Current RISC analysis fully supports this modeling style

Scaling RISC: Overcoming Obstacle 5

• New TLM-2.0: Producer-Consumer Example

- No channels! Threads operate directly in others’ modules
  - Fast, but dangerous execution in foreign territory
  - Current RISC analysis cannot handle this modeling style
Scaling RISC: Overcoming Obstacle 5

- TLM-2.0 is quite different from traditional TLM
  - Need significant extension of compiler analysis
    - Conflict analysis must follow the initiator’s threads execution through the hierarchical model structure to the targets
  - Leverage and extend existing RISC technology
    - Instance tree, instance path, and instance ID [RISCv020]
    - Hybrid analysis [ASPDAC'17]
    - Port-Call-Paths (PCP) [DATE'18]

Concluding Remarks

- On the Limits of Standard-compliant Parallel Simulation of SystemC
  - Seven Obstacles stand in the way of parallel SystemC
    - Co-routine semantics, sequential simulator state primitives, lack of thread-safety, weak role of channels, TLM-2.0, temporal decoupling, and an overall sequential modeling mindset
  - Truly parallel SystemC appears elusive given the current IEEE Standard 1666-2011
- Parallel Simulation with Maximum Compliance
  - Example: Recoding Infrastructure for SystemC (RISC)
    - Out-of-order Parallel Discrete Event Simulation
    - Dedicated SystemC compiler and parallel simulator
    - Multi- and many-core host platforms
    - Two orders of magnitude faster simulation with full accuracy
    - Open source
Acknowledgments

• For solid work, fruitful discussions, and honest feedback, I would like to thank:
  – My team at UCI
    • Zhongqi Cheng
    • Guantao Liu
    • Daniel Mendoza
    • Tim Schmidt
  – Our collaborators at Intel
    • Ajit Dingankar
    • Desmond Kirkpatrick
    • Abhijit Davare
    • Philipp Hartmann
  – And many others…
• This work has been supported in part by substantial funding from Intel Corporation. Thank you!

Selected References

• [RISC v0.4.2] RISC project, release 0.4.2: http://www.cecs.uci.edu/~doemer/risc.html