Out-of-Order Parallel Simulation of SystemC Models using the RISC Framework

Tutorial at Embedded Systems Week 2020
September 20, 2020

Rainer Dömer
doemer@uci.edu
Center for Embedded and Cyber-Physical Systems
University of California, Irvine

Virtual Tutorial Logistics

• ESWeek 2020 is a Virtual Conference
  – This tutorial is virtual, too!
• Zoom Meeting ID: 985 6461 2963
  – 9:00 AM - 1:00 PM EDT (GMT-4)
  – https://uci.zoom.us/j/98564612963
  – No meeting passcode needed

➢ Live Presentation via Zoom
  ➢ Session will be recorded for offline reference
  ▪ Presenter: Live on camera
  ▪ Audience: Please turn on your camera as well
    ➢ Request-but-not-require policy
  ➢ Interactive Discussion: Please Participate
    ➢ Chat, Raise hand, Polls, Breakout rooms, Reactions…
Poll: Time Zone

- Which time zone are you in? (Choose the closest one)
  [Single Choice]
  - Answer 1: PDT (Los Angeles) UTC-7
  - Answer 2: EDT (New York) UTC-4
  - Answer 3: CEST (Paris) UTC+2
  - Answer 4: IST (New Delhi) UTC+5:30
  - Answer 5: CST (Beijing) UTC+8
  - Answer 6: AEST (Sydney) UTC+10

Agenda

- Part 1: 9:00am EDT
  Introduction to Out-of-Order Parallel Discrete Event Simulation
- Part 2: 9:40am EDT
  Overcoming the Obstacles of IEEE SystemC Semantics
- Part 3: 10:20am EDT
  RISC: Recoding Infrastructure for SystemC
- Part 4: 11:15am EDT
  Hands-on Practical Training with RISC Compiler and Simulator
- Part 5: 12:15pm EDT
  Hands-on Practical Analysis of Parallel Potential of SystemC Models

Note: For hands-on participation, you will need a Linux account on a multi-core host with Docker access to download RISC (otherwise you are welcome to just watch the demos)
Out-of-Order Parallel Simulation
of SystemC Models
using the RISC Framework

Part 1: Introduction to
Out-of-Order Parallel Discrete Event Simulation

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 and
  – Simulation
• … of systems containing
  – Hardware and
  – Software
  ➢ Discrete Event Simulation
    – Accellera (sequential)
    – RISC (parallel)
Discrete Event Simulation (DES)

- SystemC uses Traditional 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
- Cooperative Multi-Tasking
  - IEEE 1666-2011 standard
  - A single thread is active at any time
    - Does not exploit parallelism
    - Cannot utilize multiple cores
    - Sequential simulation is slow

Specific Example:
Accellera SystemC Proof-of-Concept Library
- uses an extra root thread
  - Elaboration phase
  - Scheduling
    - Event notifications
    - Channel updates
    - Delta cycle updates
    - Simulation time updates
  - SC_METHOD calls
    - (not shown)
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]

Improved Modeling Techniques
- TLM temporal decoupling
- Savoiu et al. [MEMOCODE’05]
- Razaghi et al. [ASPDAC’12]

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’12]
- Yun et al. [TCAD’12]
- Schmidt et al. [DAC’17]
- and many others

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

Discrete Event Simulation (DES)

- Traditional DES Algorithm (sequential)
  - Active threads are managed in a READY queue
  - Waiting threads are managed in WAIT queues
    - wait(event);
    - wait(time);
  - Simulation progress
    - Delta cycle
    - Time cycle
  - Scheduler picks a single thread and executes it

start

\[
\begin{align*}
\text{if } \text{READY} = 0 & \text{ then, } n = \text{PickREADY, RUN: Go!}; \\
\text{sleep}; \\
\text{update the simulation time, move the earliest \text{WAITTOR} to \text{READY}}; \\
\text{set } \text{READY} = 0 & \text{ and end.}
\end{align*}
\]
Parallel Discrete Event Simulation (PDES)

- **Parallel DES Algorithm**
  - Active threads are managed in a READY queue
  - Waiting threads are managed in WAIT queues
  - Simulation progress
    - Delta cycle
    - Time cycle
  - Scheduler *picks N threads* and executes them *in parallel*
  - \( N \) = number of available CPU cores

![Flowchart of PDES Algorithm](image)

Parallel Discrete Event Simulation (PDES)

- **Parallel DES Algorithm requires safe synchronization**
  - Locks and condition variables (e.g. POSIX multi-threading)
  - Protected scheduling resources
  - Protected communication
  - MT-safe SystemC primitives
  - Example: Life-cycle of a SC_THREAD
Parallel Discrete Event Simulation (PDES)

- **Parallel DES [Fujimoto1990]**
  - Threads execute in parallel *iff*
    - in the same delta cycle, *and*
    - in the same time cycle
  - Significant speed up!
  - Cycle boundaries are absolute barriers: *Synchronous PDES*

- **Aggressive Asynchronous PDES**
  - Conservative Approaches
    - Careful static analysis prevents conflicts
  - Optimistic Approaches
    - Conflicts are detected and addressed *(roll back)*

- **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
  - Results in even higher speedup!
    - [DATE’12], [IEEE TCAD’14]
  - Needs compiler support for data and event *conflict analysis!*
    - Preserve the accuracy of cause and effect relationship
    - Accurate results, accurate timing
Summary and Analysis

- Traditional Discrete Event Simulation (DES)
  - Simulator runs *sequentially*, executes one thread at a time
  - Cannot utilize the parallelism of multi- or many-core hosts
- Parallel Discrete Event Simulation (PDES)
  - Threads run in *parallel* (if at the same delta and time cycle)
  - Simulation cycles are absolute barriers
- Out-of-order Parallel DE Simulation (OoO PDES)
  - Non-conflict threads run in *parallel and ahead-of-time* [DATE’12]
  - Maximum parallelism, order of magnitude speedup! [TCAD’14]

➢ Problem solved!? Not quite!
➢ What about host platforms? Readily available.
➢ What about accuracy? Achievable with careful analysis.
➢ What about standard compliance? That’s where the problem is!

Problem Definition

- What is given?
  - Embedded systems are parallel
  - SystemC is suitable and standard for system design
  - Models exhibit explicit thread-level parallelism
  - Multi- and many-core host platforms are readily available
- What do we want?
  - Fastest Parallel Discrete Event Simulation
  - For the SystemC language
- What is the objective?
  - Maximize compliance with the IEEE 1666-2011 standard
➢ Why is this so difficult?
  ➢ There are “Seven Obstacles in the Way of Standard-Compliant Parallel SystemC” [ESL’16]
Out-of-Order Parallel Simulation of SystemC Models using the RISC Framework

Part 2: Overcoming the Obstacles of IEEE SystemC Semantics

Rainer Dömer
doemer@uci.edu
Center for Embedded and Cyber-Physical Systems
University of California, Irvine

Compliance with IEEE SystemC Semantics

- IEEE Standard 1666™-2011
  - Revision of IEEE Std. 1666-2005

  ... unfortunately stands in the way of parallel SystemC simulation!

  SystemC Evolution Day 2016

  - "Seven Obstacles in the Way of Parallel SystemC Simulation",
    Rainer Doemer, Munich, Germany, May 2016.

  SystemC standard
    - ... must embrace true parallelism
    - ... must evolve in a major revision (3.x)
Poll: SystemC Evolution

- Take a guess, what happened?
  [Single Choice]
  - Answer 1: The speaker was thrown off the stage
  - Answer 2: SystemC Revolution:
    Major changes to SystemC standard
  - Answer 3: SystemC Evolution:
    Minor changes to SystemC standard
  - Answer 4: Nothing, SystemC standard didn't change.
  - Answer 5: Parallel simulation of SystemC changed.

Compliance with IEEE SystemC Semantics

- SystemC Evolution!?
  - Nothing substantial has changed…
- In absence of major changes to SystemC standard,
  my group worked hard on the compliance problem
  - “Let's make the best of it!”
  - Accept SystemC standard 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 pushes the limits to overcome the 7 obstacles!
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.

Obstacle 1: Co-Routine Semantics

• 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.”

  ```
  int x; // shared variable
  void thread1() {
    int x = 0;
    x = x + 1;
    cout << x;
  }
  void thread2() {
    int 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
  - 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.

- Proposal: Explicitly allow parallel execution, preemption
  - Process instances 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

Overcoming the Obstacles

- Obstacle 1: Resolved!
  - Introduce a dedicated SystemC Compiler
  - Automatic analysis of parallel access conflicts
  - Run SystemC processes in parallel if there are no conflicts
  - Faster simulation
  - Results remain the same
Obstacle 2: Simulator State

- **Fact:** Discrete Event Simulation (DES) is presumed
  - Example from IEEE 1666-2011, page 31: `sysc/kernel/sc_simcontext.h`

  ```c
  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
  - The general notion of shared state needs attention…

Overcoming the Obstacles

- **Obstacle 2:** Ongoing…
  - Review and revise the SystemC API
    - Slightly adjust the semantics
    - Maximize compliance with standard
  - For APIs on the slide:
    - Users’ expectations can be met
    - Example: SystemC integration with virtual platforms works fine
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

Overcoming the Obstacles

- Obstacle 3: Ongoing…
- Revise SystemC primitives for multi-thread safety
  - Protection by inserted locks
  - Store state in local or thread-local storage
  - For deterministic debugging, user can control number of parallel threads (e.g. set to 1)
Obstacle 4: Class sc_channel

• Fact: `sc_channel` is an alias type for `sc_module`
  - IEEE 1666-2011, Section “5.2.23 sc_behavior and sc_channel” (page 56):
    - The typedefs `sc_behavior` and `sc_channel` are provided for users to express their intent.
    - NOTE—There is no distinction between a behavior and a hierarchical channel other than a difference of intent. Either may include both ports and public member functions.
    - `systemc-2.3.1/include/sysc/kernel/sc_module.h`
      ```
      [...] typedef sc_module sc_channel;
      typedef sc_module sc_behavior;
      [...]```

• Problem: Alias type is only another name, no new type
  - Language does not distinguish modules and channels
    - No separation of communication and computation
      - Breaks a key system-level design principle...
  - Proposal: Class `sc_channel`, derived from `sc_module`
    - Module encapsulates computation (hosts threads/processes)
    - Channel encapsulates communication (implemented interfaces)

Overcoming the Obstacles

• Obstacle 4: Fixed!
  - Derive `sc_channel` from base class `sc_module`
  - Minimal change in SystemC headers
  - Two different types at compile-time
  - Easy distinction in static analysis
  - No known negative side-effects
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
  - Proposal: Encapsulate communication methods in channels
Overcoming Obstacle 5

- Classic TLM: Producer-Consumer Example
  - Modules wrap computation, channels wrap communication
  - Threads operate in their own modules or protected channels
  - Well-behaved execution in safe execution contexts

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
  - Requires deep analysis and well-designed models
Overcoming the Obstacles

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
  - Proposal: Encapsulate communication methods in channels
- **Socket Call Path (SCP) analysis**
- **Variable Entanglement analysis**
  - Compile-time analysis can identify target methods executed by TLM-2.0 calls
- **Support for interconnect modules and DMI**
  - [CODES+ISSS’19, ACM TECS]

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): Each thread or clocked thread process requires its own execution stack. As a result, context switching between thread processes may impose a simulation overhead when compared with method processes.
- **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, eliminate 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
Overcoming the Obstacles

- **Obstacle 6: Not a problem**
  - **SC_METHOD**, **SC_THREAD**, **SC_CTHREAD** can all be supported
  - Static analysis per process type
  - **SC_METHOD** execution by dedicated invoker threads
  - Nice optimization problem for efficient grouping with minimal conflicts

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):
    - 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.
    - 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 is one solution (fully automatic, accurate)
Overcoming the Obstacles

• Obstacle 7: Ongoing…
  ➢ Investigation needs examples
  ➢ Speed vs. accuracy tradeoff in PDES
  ➢ Out-of-order PDES can likely achieve the same benefit
  ➢ Without loss of accuracy (?)

Obstacle 7: Temporal Decoupling

• Fact: TD is designed to speed up sequential DES
  ➢ IEEE 1066-2011, Section 13.1 on "TLM-2.0 global quantum" (page 45)
  ➢ Support decoupling, possibly "simple", processes to run in parallel at different rates
  ➢ 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 sc_global_quantum, promote wait(time)
    ➢ Consider the use of a compiler to optimize scheduling, timing
    ➢ Out-of-Order PDES is one solution (fully automatic, accurate)

Summary and Analysis

• Overcoming 7 Obstacles towards Parallel SystemC
  1. Co-Routine Semantics: Resolved
  2. Simulator State: Ongoing…
  3. Lack of Thread Safety: Ongoing…
  4. Class sc_channel: Fixed
  5. TLM-2.0: Reevaluated, Resolved
  6. Sequential Mindset: Not a problem
  7. Temporal Decoupling: Ongoing…

➢ So the problem is not solved yet, but we’re getting closer!
➢ Next, let’s look at the state of the art:
  Recoding Infrastructure for SystemC (RISC)
Poll: SystemC Future

- Towards truly parallel simulation, do you expect the SystemC standard to further evolve?

  [Single Choice]
  - Answer 1: Yes, SystemC will evolve.
  - Answer 2: No, SystemC will remain as is.
  - Answer 3: No, SystemC will be replaced with another language.
  - Answer 4: Not sure, I don't know.

Out-of-Order Parallel Simulation of SystemC Models using the RISC Framework

Part 3: RISC:
Recoding Infrastructure for SystemC

Rainer Dömer
doemer@uci.edu

Center for Embedded and Cyber-Physical Systems
University of California, Irvine
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
  - BSD license
  - Thanks to Intel Corporation!

Poll: Simulator Run Time

- For your larger SystemC design models, how long is the typical simulator run time?
  [Single Choice]
  - Answer 1: A few seconds
  - Answer 2: About a minute
  - Answer 3: About 10 minutes
  - Answer 4: About an hour
  - Answer 5: Several hours
  - Answer 6: About a day
  - Answer 7: Several days.
  - Answer 8: Not sure, I don't know.
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

---

Recoding Infrastructure for SystemC (RISC)

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

  ![RISC Diagram]
  
  - ROSE Internal Representation
  - Explicit support for
    - Source code analysis
    - Source-to-source transformations

  Source: Lawrence Livermore National Laboratory (LLNL)
Recoding Infrastructure for SystemC (RISC)

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

RISC Compiler

- RISC Software Stack
  - Recoding Infrastructure for SystemC
    1) Segment Graph construction
    2) Parallel access conflict analysis

Step 1:
Build a Segment Graph (SG)
RISC Compiler

• Segment Graph Construction
  – Segment Graph (SG) 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 can be constructed statically by the compiler from the model source code
        • Let’s look into this in detail by use of a few examples!

RISC Compiler

• Segment Graph Construction
  – Example: Source code and Segment Graph

int a;
if (cond) {
  int b;
  wait(1);
  int c;
} else {
  int d;
  int e;
  wait(2);
  int f;
} while (cond) {
  int g;
}
int h;

(int a;)
(int b;)
wait(1);
(int c; int e;)
(int d;)
(int f; condition)
wait(2);
(int g; int h)
RISC Compiler

- **Segment Graph Construction**
  - Example for straight-line code

```c
void straight() {
    x = 42;
    int xx = 43;
    int yy;
    int o = y;
    wait(10, SC_NS);
    wait();
    int kk;
    wait();
    int oo;
}
```

- **Segment ID:** 0
  - input_straight.cpp:24 (this) -> x = 42
  - input_straight.cpp:25 int xx = 43;
  - input_straight.cpp:26 int yy;
  - input_straight.cpp:27 yy
  - input_straight.cpp:28 int o =(this) -> y.

- **Segment ID:** 1 (input_straight.cpp:30)
- **Segment ID:** 2 (input_straight.cpp:32)
- **Segment ID:** 3 (input_straight.cpp:37)

RISC Compiler

- **Segment Graph Construction**
  - Example for conditional control flow

```c
if_statement() {
    wait();
    int aaa;
    if(test) {
        int bbb;
        wait();
        int ccc;
    }
    int ddd;
    wait();
    int eee;
}
```

- **Segment ID:** 0
  - input_if_else.cpp:27 int test
  - input_if_else.cpp:32 int aa;
  - input_if_else.cpp:34 int bbb;
  - input_if_else.cpp:36 int ccc;
  - input_if_else.cpp:38 int ddd;
  - input_if_else.cpp:39 int eee;

- **Segment ID:** 1 (input_if_else.cpp:27)
  - input_if_else.cpp:28 int aa;
  - input_if_else.cpp:30 int bbb;
  - input_if_else.cpp:34 int ddd;

- **Segment ID:** 2 (input_if_else.cpp:31)
  - input_if_else.cpp:32 int ccc;
  - input_if_else.cpp:34 int ddd;

- **Segment ID:** 3 (input_if_else.cpp:35)
  - input_if_else.cpp:36 int eee;
RISC Compiler

• Segment Graph Construction
  – Example for repetition (loops)
    while, do-while, for
    (with break, continue)

```c
void while_continue_statement()
{
    int kk;
    while(test){
        int aa;
        wait();
        int bb;
        if(test1) {
            continue;
        } int cc;
        wait();
        int dd;
        wait();
    }
}
```

Segment ID: 3 (input_while_continue.cpp:62)
Segment ID: 0 input_while_continue.cpp:49 int kk; compilerGenerated:0 (this) -> test
Segment ID: 2 input_while_continue.cpp:58 int cc; compilerGenerated:0 (this) -> test1 input_while_continue.cpp:55 continue;
Segment ID: 1 input_while_continue.cpp:52 int bb; compilerGenerated:0 (this) -> test input_while_continue.cpp:50 int aa; input_while_continue.cpp:60 int dd;

RISC Compiler

• Segment Graph Construction
  – Example for function calls

```c
void f() int g1()
{
    int aa; int g_0;
    wait(); wait();
    int bb; int g_1 = 33;
    g1();
    if(g_1 == 88) {
        int cc; int g_2;
        wait(); wait();
        int dd; int g_3 = 44;
        return 43;
        int DEAD_CODE;
    }
    int g_4;
    wait();
    int g_5;
    wait();
    int g_6;
    int return_value = 2;
    return return_value;
}
```

Segment ID: 1 input_function_calls.cpp:148 int aa;
Segment ID: 0 input_function_calls.cpp:144 int bb; g1();
Segment ID: 2 input_function_calls.cpp:164 int g_1 = 33;
Segment ID: 3 input_function_calls.cpp:170 int g_2;
Segment ID: 4 input_function_calls.cpp:175 return_value = 2;
Segment ID: 5 input_function_calls.cpp:180 return_value;

(c) 2020 R. Doemer et al., CECS
RISC Compiler

- Segment Graph Construction
  - Example for recursive function calls
    - Direct, indirect recursion

```c
void main()
{
    wait();
    f();
    wait();
}
```

```c
void f()
{
    wait();
    if(xx>0) {
        wait();
        g();
        wait();
    }
}
```

```c
void g()
{
    xx--; wait();
    if(xx>0) {
        wait();
        int before_rec;
        f();
        int after_rec;
        wait();
    } else {
        wait();
        return;
    }
}
```

Tutorial at ESWEEK, Sept. 20, 2020

RISC Compiler

- Parallel Access Conflict Analysis for Segments
  - 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

![Conflict Segments Diagram](image)
RISC Compiler

- **Parallel Access Conflict Analysis for Segments**
  - Variable analysis for Read, Write, and Read/Write accesses
  - Example:

    ```cpp
    class Conflict : public sc_module {
      SC_CTOR(Conflict) {
        SC_THREAD(thread1);
        SC_THREAD(thread2);
      }
      int x, y, z;
    
    void thread1() {
      int a;
      a = 2;
      wait();
      a = x + y;
      wait();
      z++;
    }
    
    void thread2() {
      int b = 2;
      x = y;
      wait();
      x = y * z;
      wait();
      z++;
    }
    }
    ```

    Segment Graph

    Segment ID: 0
    `conflict.cpp:24 int a;`
    `conflict.cpp:25 a = 2`

    Segment ID: 3
    `conflict.cpp:34 int b = 2;`
    `conflict.cpp:35 x = y`

    Segment ID: 4 (conflict.cpp:36)
    `conflict.cpp:37 x = y * z`
    `conflict.cpp:39 z++`

    Segment ID: 5 (conflict.cpp:38)
    `conflict.cpp:40 x = y * z`
    `conflict.cpp:41 z++`

    Segment ID: 6 (conflict.cpp:40)
    `conflict.cpp:41 z++`
RISC Compiler

- Parallel Access Conflict Analysis for Segments
  - Variable analysis for Read, Write, and Read/Write accesses
  - Example:

```
Variable Accesses

Segment ID: 0
- (W) a
- (R) x

Segment ID: 1 (par/1.cpp:26)
- (R) x

Segment ID: 2 (par/2.cpp:30)
- (W) x
- (R) z

Segment ID: 3
- (W) y

Segment ID: 4 (par/4.cpp:36)
- (R) x

Segment ID: 5 (par/5.cpp:40)
- (W) x
- (R) a

Segment ID: 6 (par/6.cpp:44)
- (W) x

Data Conflict Table

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

RISC Compiler

- RISC Software Stack
  - Recoding Infrastructure for SystemC
    1) Segment Graph construction
    2) Parallel access conflict analysis
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

RISC Simulator

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

- Issue threads...
  - truly in parallel and out-of-order
  - whenever they are ready
  - and have no conflicts!
  - Fast conflict table lookup
  - Optimized thread-to-core mapping
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</th>
<th>PDES</th>
<th>OOO PDES</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Run Time</td>
<td>CPU Load</td>
<td>Run Time</td>
</tr>
<tr>
<td>1</td>
<td>162.13 s</td>
<td>99%</td>
<td>162.06 s</td>
</tr>
<tr>
<td>2</td>
<td>162.19 s</td>
<td>99%</td>
<td>96.50 s</td>
</tr>
<tr>
<td>4</td>
<td>162.56 s</td>
<td>99%</td>
<td>54.00 s</td>
</tr>
<tr>
<td>8</td>
<td>163.10 s</td>
<td>99%</td>
<td>29.89 s</td>
</tr>
<tr>
<td>16</td>
<td>164.01 s</td>
<td>99%</td>
<td>19.03 s</td>
</tr>
<tr>
<td>32</td>
<td>165.89 s</td>
<td>99%</td>
<td>11.78 s</td>
</tr>
<tr>
<td>64</td>
<td>170.32 s</td>
<td>99%</td>
<td>9.79 s</td>
</tr>
<tr>
<td>128</td>
<td>174.55 s</td>
<td>99%</td>
<td>9.34 s</td>
</tr>
<tr>
<td>256</td>
<td>185.47 s</td>
<td>100%</td>
<td>8.91 s</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!

Experiments and Results

- Parallel Benchmark Results (Xeon Phi Coprocessor, 60x4 cores)

Execution Time [sec] vs. Speedup
Conclusion

- Recoding Infrastructure for SystemC (RISC)
  - Out-of-Order 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 use and collaboration (BSD license)
    - Thanks to Intel Corporation!

Acknowledgments

- For solid work, fruitful discussions, and honest feedback, I would like to thank:
  - My team at UCI
    - Emad Arasteh, Aditya Harit, Vivek Govindasamy
    - Zhongqi Cheng, Daniel Mendoza
    - Tim Schmidt, Guantao Liu
    - Farah Arabi, Spencer Kam
  - 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!
Out-of-Order Parallel Simulation of SystemC Models using the RISC Framework

Part 4: Hands-on Practical Training with RISC Compiler and Simulator

Rainer Dömer
doemer@uci.edu

Center for Embedded and Cyber-Physical Systems
University of California, Irvine

RISC Open Source Releases

RISC Open Source Releases

- RISC Compiler and Simulator, Release V0.6.2
  - [http://www.cecs.uci.edu/~doemer/risc.html#RISC062](http://www.cecs.uci.edu/~doemer/risc.html#RISC062)
  - Installation notes and script: INSTALL, Makefile
  - Open source tar ball: risc_v0.6.2.tar.gz
  - Docker script and container: Dockerfile
  - Doxygen documentation: RISC API, OOPSC API
  - Tool manual pages: risc, visual, sysdot ...
  - BSD license terms: LICENSE
- Companion Technical Report

- Docker container:
  - bash# docker pull ucirvinelecs/risc062
  - bash# docker run -it ucirvinelecs/risc062
  - [dockeruser]# cd demodir
  - [dockeruser]# make play_demo
  - [dockeruser]# https://hub.docker.com/r/ucirvinelecs/risc062/

RISC Compiler Tool Flow

- Compile and simulate with Accellera:
  - g++ play.cc ...
  - ./play_seq
- Compile and simulate with RISC:
  - risc play.cc ...
  - ./play_ooo
- Measure simulator run time:
  - /usr/bin/time ...

Traditional SystemC Simulation

- SystemC Model
- Header File
- C++ Compiler
- Executable
- Host PC
  - Sequential (1x)

RISC OoO Parallel Simulation

- SystemC Model
- Header File
- RISC Compiler
- Parallel Executable
- Multi-/Many-Core Host
  - Out-of-order Parallel (10x – 100x)
Demo Example 1

- Conceptual DVD Player, TLM-1.0 style
  - Parallel video and audio decoding with different frame rates

```plaintext
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:   }  
11: };
```

```plaintext
1: SC_MODULE(AudioCodec)  
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(26120, SC_US);  
9:     p2->send(outFrm);  
10:   }  
11: };
```

1. Real time schedule: fully parallel

2. Reference simulator schedule (DES)
Demo Example 1

- Conceptual DVD Player, TLM-1.0 style
  - Parallel video and audio decoding with different frame rates
    1. Real time schedule: fully parallel
    3. Synchronous parallel schedule (PDES)
    4. Out-of-order parallel schedule (OoO PDES)

(c) 2020 R. Doemer et al., CECS
Demo Example 1

- Conceptual DVD Player, TLM-1.0 style
  - 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>

Demo Example 2

- Conceptual DVD Player, TLM-2.0 style
  - Example: hierarchical socket binding, event handshakes
Demo Example 2

- Various Modeling Styles Supported by RISC v0.6.2
  - Structural Composition
  - Synchronization
  - Connectivity
  - Explicit Memories
  - Interconnect Modules
  - DMI

Experimental Results for TLM-2.0 DVD Player Models
- All models are functional and simulate correctly (RISC v0.6.0)
- Results: run time (seconds) and speedup (%)

<table>
<thead>
<tr>
<th>Interface</th>
<th>Direct</th>
<th>Hierarchical</th>
<th>Interconnect</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Seq OoO Par</td>
<td>Seq OoO Par</td>
<td>Seq OoO Par</td>
</tr>
<tr>
<td>BTI</td>
<td>208.1 73.8 282%</td>
<td>208.1 75.7 274%</td>
<td>208.4 74.8 278%</td>
</tr>
<tr>
<td>DMI</td>
<td>208.2 73.7 282%</td>
<td>208.5 75.5 276%</td>
<td>208.4 74.7 279%</td>
</tr>
<tr>
<td>NBTL</td>
<td>209.3 74.9 279%</td>
<td>209.4 75.6 277%</td>
<td>209.5 75.7 277%</td>
</tr>
</tbody>
</table>

- All models exhibit high simulation speedup
  - 2.8 times faster than the sequential reference model
  - This beats our prior results: TLM-1.0 reached only 2.5 x
Demo Example 3

• 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

Demo Example 3

• Experimental Results
  – 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>8.62 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>

(c) 2020 R. Doemer et al., CECS
Interactive Hands-on Demo

• Docker Setup
  ➢ `sudo docker pull ucirvinelecs/risc062`
  ➢ `sudo docker run -it ucirvinelecs/risc062:latest`
  ➢ `cd demodir`
• Demo 1: DVD Player, TLM-1.0
  ➢ `make play_demo`
• Demo 2: DVD Player, TLM-2.0
  ➢ `make play_TLM2_bus_mem_demo`
• Demo 3: Mandelbrot Renderer
  ➢ `make mandelbrot_demo`
• Handout and detailed “Cheat Sheet” available online
  ➢ `http://www.cecs.uci.edu/~doemer/ESWeekTutorial.txt`

Out-of-Order Parallel Simulation of SystemC Models using the RISC Framework

Part 5: Hands-on Practical Analysis of Parallel Potential of SystemC Models

Rainer Dömer
doemer@uci.edu

Center for Embedded and Cyber-Physical Systems
University of California, Irvine
RISC Framework Overview

- RISC Framework consists of 3 Branches
  1. Simulation
     - With VP support
  2. Analysis
     - Static
     - Dynamic
  3. Recoding
     - Transformation
     - Optimization

This session demonstrates SystemC analysis features

Analysis of Model Structure

- SystemC Model Hierarchy
- SystemC Model Connectivity
  - Ports
  - Sockets

(c) 2020 R. Doemer et al., CECS
Visualization of Model Structure

- **SystemC Model Visualization:** **visual**
  - Hierarchy and connectivity
    - Ports and sockets
  - Threads in modules

Analysis of Potential Parallelism

- **Segment Graph based Conflict Analysis**
  1. Build the Segment Graph
  2. Perform parallel access conflict analysis
  3. Instrument the model for parallel execution
Visualization of Segment Graph

- Segment Graph and Conflicts Visualization: 
  `sysdot`

Visualization of Conflict Tables

- Web browser and `sysdot`
Interactive Hands-on Demo

- Docker Setup
  - `xhost +`
  - `sudo docker run -it --net=host --env="DISPLAY"`
  - `-volume="$HOME/.Xauthority:/root/.Xauthority:rw"`
  - `ucirvinelecs/risc062:latest`
  - `cd demodir`
- Demo 5: Examples using `visual`
  - `visual play.cpp`
  - (and other examples)
- Demo 6: Examples using `sysdot`
  - `make play_ooo`
  - `sysdot play_segment_graph.dot`
  - (and other examples)
- Handout and detailed “Cheat Sheet” available online
  - `http://www.cecs.uci.edu/~doemer/ESWeekTutorial.txt`

References (1)

References (2)


References (3)