Towards Parallel Simulation of Multi-Domain System Models

Keynote
DAC 2015 Workshop
System-to-Silicon Performance Modeling and Analysis

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

Towards Parallel Simulation…

• System Modeling
  – Models
  – Domains
  – Dimensions

• System Simulation
  – Discrete Event Simulation (DES)
  – Parallel Discrete Event Simulation (PDES)
  – Out-of-Order Parallel Discrete Event Simulation (OoO PDES)

• Parallel SystemC Project
  – Dependency Analysis using Segment Graphs
  – Early Experimental Results

• Concluding Remarks
System Modeling

• What is a Model?
  – A model is an abstraction of reality.

• What is Abstraction?
  – Abstraction is the intentional omission or simplification of details.
    ➢ Reduces system complexity!
    ➢ Allows to focus on the essentials!

• What is Modeling?
  – Creating and shaping a model:
    • Choosing and including properties of interest
    • Simplifying of characteristics of limited interest
    • Omitting of aspects of no interest

• Our Interest
  – Cyber-Physical and Embedded Systems

System Modeling Domains

• Traditional Properties of Interest in System Design
  – Functionality (execution)
    • Computation of functional result (data values)
    • Validation of correctness, Boolean value
  – Performance (estimation of speed, meeting deadlines)
    • Execution time \( t \) [s], optimization goal
  – Structure (number and type of components, estimation of cost,)
    • Chip area \([\text{mm}^2]\), optimization goal

• New Domains of Increasing Importance
  – Energy consumption (i.e. battery run-time)
    • Power \( P \) [W] = \( E \) [J] / \( t \) [s], optimization goal with threshold
  – Thermal behavior (need for cooling)
    • Temperature \( T \) [°C], threshold value, \( T < T_{\text{critical}} \)
  – Reliability, degradation, aging
    • Mean time between failures MTBF [s], optimization goal
  – ...

(c) 2015 R. Doemer, CECS
System Modeling Domains

- Traditional Properties of Interest in System Design
  - **Functionality** (execution)
    - Computation of functional result (data values)
    - Validation of correctness, Boolean value
  - **Performance** (estimation of speed, meeting deadlines)
    - Execution time $t \text{ [s]}$, optimization goal
  - **Structure** (number and type of components, estimation of cost)
    - Chip area $\text{ [mm}^2\text{]}$, optimization goal
- New Domains of Increasing Importance
  - **Energy consumption** (i.e. battery run-time)
    - Power $P \text{ [W]} = E \text{ [J]} / t \text{ [s]}$, optimization goal with threshold
  - **Thermal behavior** (need for cooling)
    - Temperature $T \text{ [°C]}$, threshold value, $T < T_{\text{critical}}$
  - **Reliability, degradation, aging**
    - Mean time between failures $\text{MTBF [s]}$, optimization goal
  - ...

**Extra-Functional Properties**
(e.g. [Ramesh'12], [Hartmann'15])

System Modeling Dimensions

- Traditional Properties
  - Functionality (check!)
  - Performance (max!)
  - Cost (min!)
- New Properties
  - Power (min!)
  - Thermal effects (min!)
  - Reliability (max!)
  - Degradation, Aging
  - ...

- Properties in different dimensions can be modeled independently!
- Opportunity to exploit dimensions for parallel simulation!
System Modeling Dimensions

- SystemC Example:
  - Function
  - Structure
  - Timing
  - Power consumption
  - Thermal behavior
  - Reliability degradation

SC_MODULE(Component)
{ sc_port<i_data> DataIn, DataOut;
  SCCTOR(Component)
  { SC_THREAD(main);
    }
  void main()
  { ... 
    x = f(DataIn->read());
    wait(delay1(), SC_NS);
    consume(power1(), mW);
    dissipate(temp1(), degC);
    age(mtbf1(), SC_SEC);
    ... 
    while(cond(x))
    { ... 
      y = g(x);
      wait(delay2(), SC_NS);
      consume(power2(), mW);
      dissipate(temp2(), degC);
      age(mtbf2(), SC_SEC);
    }
    DataOut->write(y);
  }
};

System Simulation

- Traditional Discrete Event Simulation
  - well-suited for functional validation and performance vs. cost trade-offs
  - but cannot cope with the additional complexities and extra-functional properties of new domains
    - Growing complexity impacts execution speed!
  - True system simulation must advance to
    1) efficiently integrate new domains and dimensions
    2) fully exploit parallel execution
    - Utilize parallelism to ensure scalability!

- Example project:
  - Out-of-Order Parallel Simulation of SystemC Models
  - Parallel, fast, and accurate
Parallel Simulation, Key Points

- Project on Advanced Parallel SystemC Simulation
  - Out-of-Order PDES on many-core host platforms
  - Maximum compliance with current execution semantics
  - Supported with funding by Intel® Corp.
- Introduction of a Dedicated SystemC Compiler
  - Recoding Infrastructure for SystemC (RISC)
  - Advanced static analysis for parallel execution
  - Model instrumentation and code generation
- Parallel SystemC Core Library
  - Out-of-order parallel scheduler, multi-thread safe primitives
  - Many-core target platform (e.g. Intel® Xeon Phi™)
- Open Source
  - Collaboration with Accellera SystemC Language WG

Discrete Event Simulation

- Traditional Discrete Event Simulation (DES)
  - Reference simulators run sequentially, only one thread at a time (cooperative multi-threading model)
  - Cannot utilize the capabilities of multi- or many-core hosts
- Parallel Discrete Event Simulation (PDES)
  - Threads run in parallel (if at the same delta cycle and time)
  - Simulation-cycles are absolute barriers!
- Out-of-order Parallel DE Simulation (OoO PDES)
  - Threads run in parallel and out-of-order [DATE’12]
    even in different delta and time cycles if there are no conflicts!
  - Aggressive, runs maximum number of threads in parallel, but fully preserves DES semantics and model accuracy!
Towards Parallel Simulation of Multi-Domain System Models

Discrete Event Simulation (DES)

- 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

- IEEE Standard Simulator
  - SystemC reference simulator
  - Uses cooperative multi-threading
  - A single thread is active at any time!
  - Cannot exploit parallelism
  - Cannot utilize multiple cores

Parallel Discrete Event Simulation (PDES)

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

- Aggressive Parallel DES
  - Conservative Approaches
    - Careful static analysis prevents conflicts
  - Optimistic Approaches
    - Conflicts are detected and addressed (roll back)
Out-of-Order Parallel DES

- **Out-of-Order PDES**
  - Threads execute in parallel *iff*
    - in the same delta cycle, *and*
    - in the same time cycle,
    - *OR if there are no conflicts!*
  - Can utilize advanced compiler for static data conflict analysis
    - Allows as many threads in parallel as possible
    - Significantly higher speedup!
      - Results at [DATE'12], [IEEE TCAD14]
    - Fully preserves...
      - DES execution semantics
      - Accuracy in results and timing
- **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)
  - Journal publication: “OoO PDES for TLM” [IEEE TCAD’14]
    - Comprehensive article with HybridThreads extension

Out-of-Order PDES Technology

- **OoO 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 on many cores
    - 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)
  - Journal publication: “OoO PDES for TLM” [IEEE TCAD’14]
    - Comprehensive article with HybridThreads extension
Project Overview and Tool Flow

- Research and Development Tasks
  1) Dedicated SystemC compiler (RISC infrastructure)
  2) Parallel SystemC headers and library
  3) Performance tuning for many-core hosts
  4) Virtual Platform (VP) integration
  5) Model analysis (may-happen-in-parallel, MHP)
  6) Model recoding, transformation and optimization

R&D Task 1: SystemC Compiler

- RISC Software Stack
  - Recoding Infrastructure for SystemC
    - C/C++ foundation
    - ROSE compiler infrastructure

  - ROSE Internal Representation
  - Explicit support for
    - Source code analysis
    - Source-to-source transformations
R&D Task 1: SystemC Compiler

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

- **Class hierarchy to represent SystemC objects**
  - RISC Software Stack
  - SystemC IR
    - ROSE IR
    - C/C++ Foundation

**Segment Graph Construction**

1. **Segment Graph construction**
2. **Segment conflict analysis**

**Model.cpp**

- **SystemC Model**
  - **SystemC Compiler**
    - RISC
      - **Segment Graph Construction**
      - **Parallel Access Conflict Analysis**

**Parallel C++ Model**

- **Compilation, Simulation**

**SystemC Model**

- **Model.cpp**

**Segment Graph**

- **Seg 1**
  - R: a, b
  - W: x
  - RW: z

- **Seg 2**
  - R: a, b
  - W: x, y
  - RW: z

- **Seg 3**
  - R: a, b
  - W: x, y
  - RW: z

**Conflicts**

- **Seg 1**
  - True
- **Seg 2**
  - True
- **Seg 3**
  - True
R&D Task 1: SystemC Compiler

- Segment Graph Construction
  - *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 can be constructed statically by the compiler from the model source code
    - (see example on next slide)
R&D Task 1: SystemC Compiler

• Segment Graph Construction: Current Status
  – Support 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)

R&D Task 1: SystemC Compiler

• Segment Graph Construction: Current Status
  – Support for conditional statements
    ➢ if, if-else, switch-case (with break)

```c
void 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:28 int aaa;
input_if_else.cpp:30 int bbb;
input_if_else.cpp:34 int ddd;

Segment ID: 1 (input_if_else.cpp:27);
compilerGenerated:0 (this) -> test
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;
R&D Task 1: SystemC Compiler

- Segment Graph Construction: Current Status
  - Support for loop statements
    - `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;
    }
}
```

Segment ID: 3 (input_while_continue.cpp:62)
Segment ID: 0 (input_while_continue.cpp:49)
Segment ID: 2 (input_while_continue.cpp:58)
Segment ID: 1 (input_while_continue.cpp:52)
Segment ID: 5 (input_function_calls.cpp:176)
Segment ID: 4 (input_function_calls.cpp:174)
Segment ID: 6 (input_function_calls.cpp:157)
Segment ID: 7 (input_function_calls.cpp:173)
Segment ID: 8 (input_function_calls.cpp:170)
Segment ID: 9 (input_function_calls.cpp:168)
Segment ID: 10 (input_function_calls.cpp:166)
Segment ID: 11 (input_function_calls.cpp:164)
Segment ID: 12 (input_function_calls.cpp:162)
Segment ID: 13 (input_function_calls.cpp:160)
Segment ID: 14 (input_function_calls.cpp:158)
Segment ID: 15 (input_function_calls.cpp:156)
Segment ID: 16 (input_function_calls.cpp:154)
Segment ID: 17 (input_function_calls.cpp:152)
Segment ID: 18 (input_function_calls.cpp:150)
Segment ID: 19 (input_function_calls.cpp:148)
Segment ID: 20 (input_function_calls.cpp:146)
Segment ID: 21 (input_function_calls.cpp:144)
Segment ID: 22 (input_function_calls.cpp:142)
Segment ID: 23 (input_function_calls.cpp:140)
Segment ID: 24 (input_function_calls.cpp:138)
Segment ID: 25 (input_function_calls.cpp:136)
Segment ID: 26 (input_function_calls.cpp:134)
Segment ID: 27 (input_function_calls.cpp:132)
Segment ID: 28 (input_function_calls.cpp:130)
Segment ID: 29 (input_function_calls.cpp:128)
Segment ID: 30 (input_function_calls.cpp:126)
Segment ID: 31 (input_function_calls.cpp:124)
Segment ID: 32 (input_function_calls.cpp:122)
Segment ID: 33 (input_function_calls.cpp:120)
Segment ID: 34 (input_function_calls.cpp:118)
Segment ID: 35 (input_function_calls.cpp:116)
Segment ID: 36 (input_function_calls.cpp:114)
Segment ID: 37 (input_function_calls.cpp:112)
Segment ID: 38 (input_function_calls.cpp:110)
Segment ID: 39 (input_function_calls.cpp:108)
Segment ID: 40 (input_function_calls.cpp:106)
Segment ID: 41 (input_function_calls.cpp:104)
Segment ID: 42 (input_function_calls.cpp:102)
Segment ID: 43 (input_function_calls.cpp:100)
Segment ID: 44 (input_function_calls.cpp:98)
Segment ID: 45 (input_function_calls.cpp:96)
Segment ID: 46 (input_function_calls.cpp:94)
Segment ID: 47 (input_function_calls.cpp:92)
Segment ID: 48 (input_function_calls.cpp:90)
Segment ID: 49 (input_function_calls.cpp:88)
Segment ID: 50 (input_function_calls.cpp:86)
Segment ID: 51 (input_function_calls.cpp:84)
Segment ID: 52 (input_function_calls.cpp:82)
Segment ID: 53 (input_function_calls.cpp:80)
Segment ID: 54 (input_function_calls.cpp:78)
Segment ID: 55 (input_function_calls.cpp:76)
Segment ID: 56 (input_function_calls.cpp:74)
Segment ID: 57 (input_function_calls.cpp:72)
Segment ID: 58 (input_function_calls.cpp:70)
Segment ID: 59 (input_function_calls.cpp:68)
Segment ID: 60 (input_function_calls.cpp:66)
Segment ID: 61 (input_function_calls.cpp:64)
Segment ID: 62 (input_function_calls.cpp:62)
Segment ID: 63 (input_function_calls.cpp:60)
Segment ID: 64 (input_function_calls.cpp:58)
Segment ID: 65 (input_function_calls.cpp:56)
Segment ID: 66 (input_function_calls.cpp:54)
Segment ID: 67 (input_function_calls.cpp:52)
Segment ID: 68 (input_function_calls.cpp:50)
Segment ID: 69 (input_function_calls.cpp:48)
Segment ID: 70 (input_function_calls.cpp:46)
Segment ID: 71 (input_function_calls.cpp:44)
Segment ID: 72 (input_function_calls.cpp:42)
Segment ID: 73 (input_function_calls.cpp:40)
Segment ID: 74 (input_function_calls.cpp:38)
Segment ID: 75 (input_function_calls.cpp:36)
Segment ID: 76 (input_function_calls.cpp:34)
Segment ID: 77 (input_function_calls.cpp:32)
Segment ID: 78 (input_function_calls.cpp:30)
Segment ID: 79 (input_function_calls.cpp:28)
Segment ID: 80 (input_function_calls.cpp:26)
Segment ID: 81 (input_function_calls.cpp:24)
Segment ID: 82 (input_function_calls.cpp:22)
Segment ID: 83 (input_function_calls.cpp:20)
Segment ID: 84 (input_function_calls.cpp:18)
Segment ID: 85 (input_function_calls.cpp:16)
Segment ID: 86 (input_function_calls.cpp:14)
Segment ID: 87 (input_function_calls.cpp:12)
Segment ID: 88 (input_function_calls.cpp:10)
Segment ID: 89 (input_function_calls.cpp:8)
Segment ID: 90 (input_function_calls.cpp:6)
Segment ID: 91 (input_function_calls.cpp:4)
Segment ID: 92 (input_function_calls.cpp:2)
Segment ID: 93 (input_function_calls.cpp:0)
```
R&D Task 1: SystemC Compiler

- Segment Graph Construction: Current Status
  - Support for recursive function calls
    
    ```c
    void main()
    {
        f();
        wait();
    }
    
    void f()
    {
        if(xx>0) {
            g();
            wait();
        }
        wait();
        return;
    }
    
    void g()
    {
        xx--;
        wait();
        if(xx>0) {
            f();
            wait();
            int before_rec;
            f();
            wait();
            int after_rec;
            wait();
        } else {
            wait();
            return;
        }
    }
    
    Segment ID: 0
    Segment ID: 1 input_recursive.cpp:151
    input_recursive.cpp:152 (this) -> recursive1();
    
    Segment ID: 2 input_recursive.cpp:159
    Segment ID: 3 input_recursive.cpp:161
    input_recursive.cpp:162 (this) -> recursive2();
    
    Segment ID: 4 input_recursive.cpp:171
    input_recursive.cpp:172 int after_rec;
    
    Segment ID: 5 input_recursive.cpp:180
    compilerGenerated 0
    
    Segment ID: 6 input_recursive.cpp:174
    input_recursive.cpp:175 int before_rec;
    
    Segment ID: 7 (input_recursive.cpp:163)
    
    Segment ID: 8 (input_recursive.cpp:165)
    compilerGenerated 0
    ```

- 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
R&D Task 1: SystemC Compiler

• Segment Conflict Analysis: Current Status
  – Variable access analysis for Read, Write, and Read/Write
  – Example:

```
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++;
    x++;
  }
```

Segment Graph
R&D Task 1: SystemC Compiler

- Segment Conflict Analysis: Current Status
  - Variable access analysis for Read, Write, and Read/Write
  - Example:

```
<table>
<thead>
<tr>
<th>Segment ID</th>
<th>Accesses</th>
<th>Segment Data Conflict Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(x, x)</td>
<td>X</td>
</tr>
<tr>
<td>2</td>
<td>(x, x)</td>
<td>X</td>
</tr>
<tr>
<td>3</td>
<td>(x, x)</td>
<td>X</td>
</tr>
<tr>
<td>4</td>
<td>(x, x)</td>
<td>X</td>
</tr>
</tbody>
</table>
```

R&D Task 2: Parallel SystemC Library

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

- Fast conflict table lookup
- Truly parallel threads
- Optimal thread-to-core mapping
R&D Task 2: Parallel SystemC Library

- Inter-Thread Communication Protection
  - Need to comply with SystemC LRM [IEEE Std 1666™]
    • Cooperative (or co-routine) multitasking semantics
      – Execution “without interruption”
    - Must protect inter-thread communication in channels!
      ➢ Insert a mutex lock into channel instances
        – To lock the channel on thread entry
        – To unlock the channel on thread exit

Channel

Thread 1

Thread 2

R&D Task 2: Parallel SystemC Library

- Inter-Thread Communication Protection
  - Need to comply with SystemC LRM [IEEE Std 1666™]
    • Cooperative (or co-routine) multitasking semantics
      – Execution “without interruption”
    - Must protect inter-thread communication in channels!
      ➢ Primitive SystemC channels
        ➢ Static protection (parallel SystemC headers, library)
      • User-defined hierarchical channels
        ➢ Dynamic protection through source code instrumentation
      ➢ Design Flow with Model Instrumentation by SystemC compiler
Early Experimental Results

• Intel® Many Integrated Core Architecture
  ➢ Intel® Xeon Phi™ Coprocessor
    – Provides
      • 60 processor cores
      • 4 hyper-threads per core
      ➢ 240 parallel hardware threads!
    – Hardware Features
      • Vector processing unit (VPU)
      • Extended Math Unit (EMU) for transcendental operations
      • Bidirectional ring interconnect
    – Peak performance
      ➢ over 1 teraFLOPS (double-precision)
      ➢ Uses familiar and standard programming models
      ➢ Appears as a regular Linux machine with 240 cores!

Early Experimental Results

• Graphics Application: Mandelbrot Set Renderer
  – Experimental Results
    • Sequence of 100 Mandelbrot images (640x448, depth 4096)
    • Manually created PDES model (Posix-threads based)
    • Multi-core platforms: Intel® Xeon® CPUs (4 cores, 2x6 cores)
    • Many-core platform: Intel® Xeon Phi™ (60 x 4 cores)

Mandelbrot Benchmark Results on Intel® Xeon® and Xeon Phi™

4 Core Host, PDES 3.7x speedup
2 CPU 6 Core Host, PDES 5.9x speedup
2 CPU 6 Core Host, OoO PDES 6.3x speedup
60x4 Core Xeon Phi, Posix PDES 46x speedup

3.7 to 6.3x
46x

**Early Experimental Results**

- Experimental Results (Intel® Xeon Phi Coprocessor, 60x4 cores)

**Parallel Benchmark Results on Intel® Xeon Phi™**

**Concluding Remarks**

- Project on Advanced Parallel SystemC Simulation
  - Out-of-Order PDES on many-core host platforms
  - Maximum compliance with current execution semantics
  - Supported with funding by Intel® Corp.

- Introduction of a Dedicated SystemC Compiler
  - Recoding Infrastructure for SystemC (RISC)
  - Advanced static analysis for parallel execution
  - Model instrumentation, code optimization, transformation

- Parallel SystemC Core Library
  - Out-of-order parallel scheduler, multi-thread safe primitives
  - Many-core target platform (e.g. Intel® Xeon Phi™)

- Open Source
  - Collaboration with Accellera SystemC Language WG
Conclusion

• System simulation must
  1) integrate new domains for extra-functional properties
     – Performance
     – Structure
     – Energy consumption
     – Thermal behavior
     – Reliability, degradation, aging
  2) exploit parallel execution to ensure scalability

• Independent domains form new dimensions
  ➢ Exploit independent dimensions for parallel simulation

References