Transaction-Level Models of Systems-on-a-Chip
Can they be Fast, Correct and Faithful?

Matthieu Moy

Verimag (Grenoble INP)
Grenoble, France

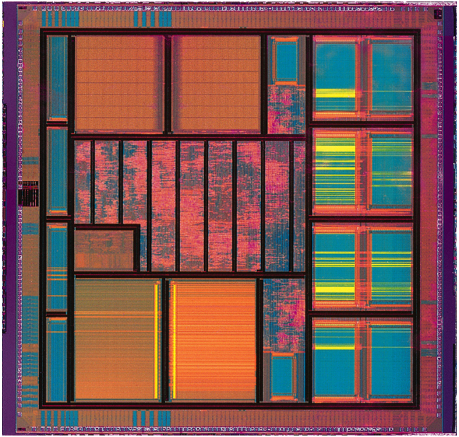
September 2012

Outline

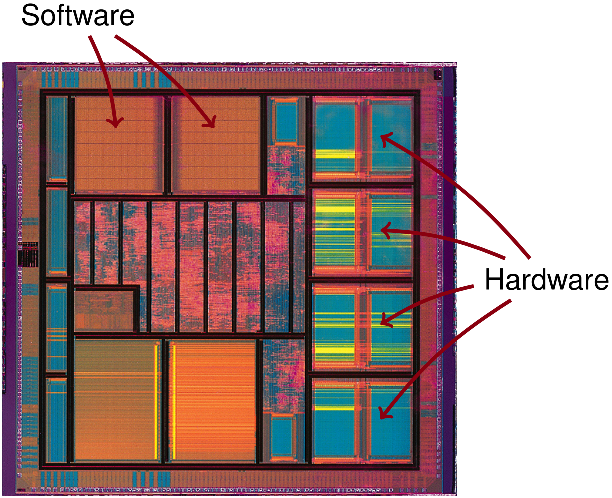
1. Introduction: Systems-on-a-Chip, Transaction-Level Modeling
2. Compilation of SystemC/TLM
3. Verification of SystemC/TLM
4. Non-functional Properties in TLM
5. Conclusion

Outline

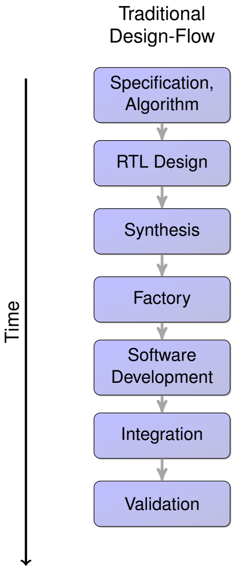
Modern Systems-on-a-Chip



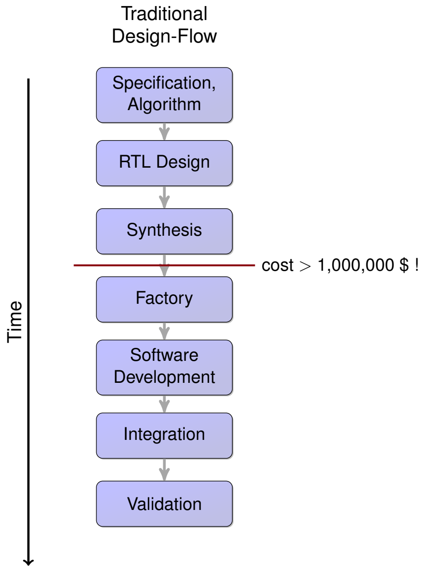
Modern Systems-on-a-Chip



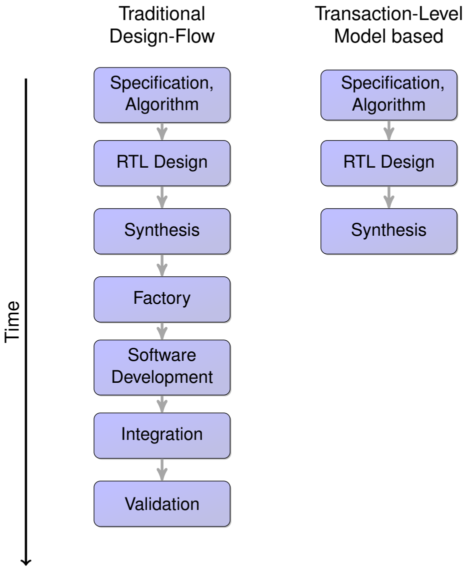
Hardware/Software Design Flow



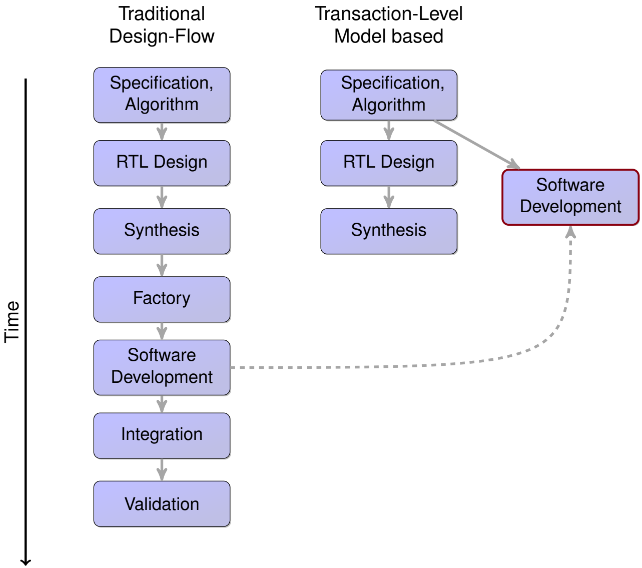
Hardware/Software Design Flow



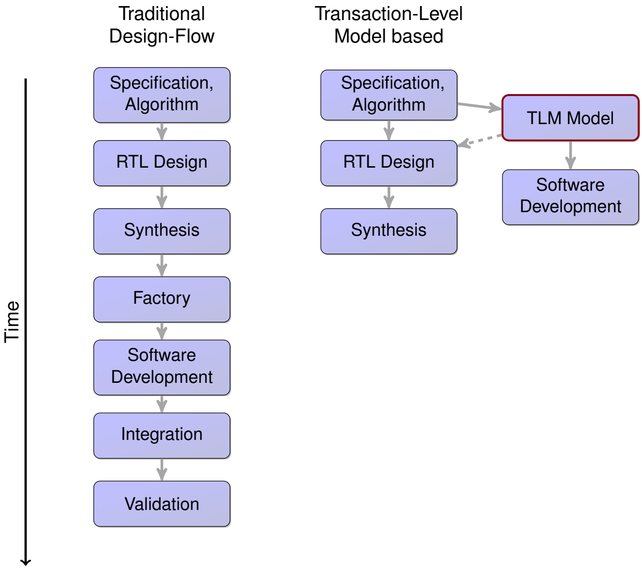
Hardware/Software Design Flow



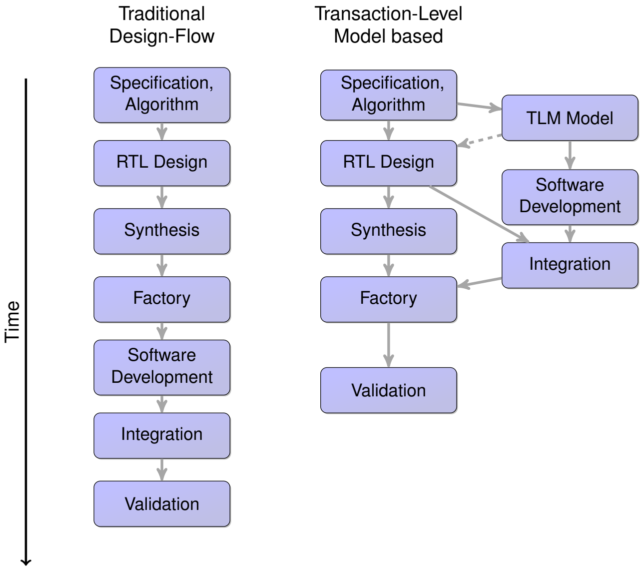
Hardware/Software Design Flow



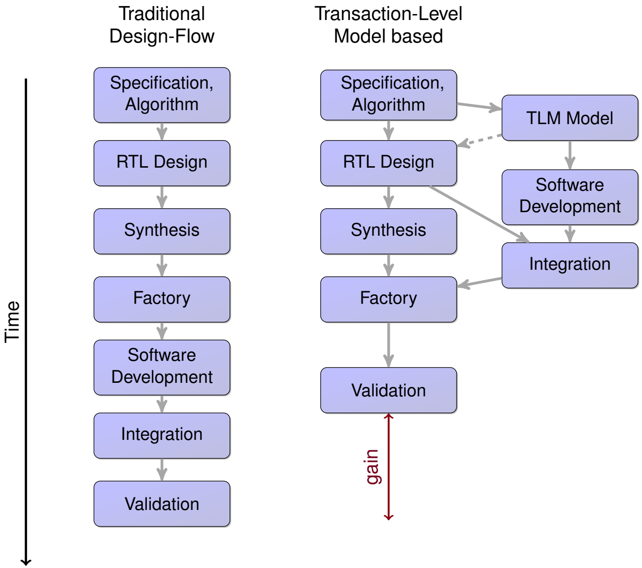
Hardware/Software Design Flow



Hardware/Software Design Flow



Hardware/Software Design Flow



The Transaction Level Model: Principles and Objectives

A high level of abstraction,
that appears early in the design-flow

The Transaction Level Model: Principles and Objectives

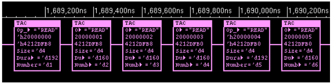
A high level of abstraction, that appears early in the design-flow

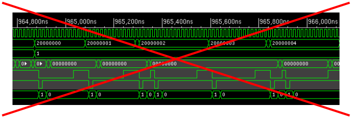
- A virtual prototype of the system, to enable
  - Early software development
  - Integration of components
  - Architecture exploration
  - Reference model for validation
- Abstract implementation details from RTL
  - Fast simulation ($\simeq$ 1000x faster than RTL)
  - Lightweight modeling effort ($\simeq$ 10x less than RTL)

Content of a TLM Model

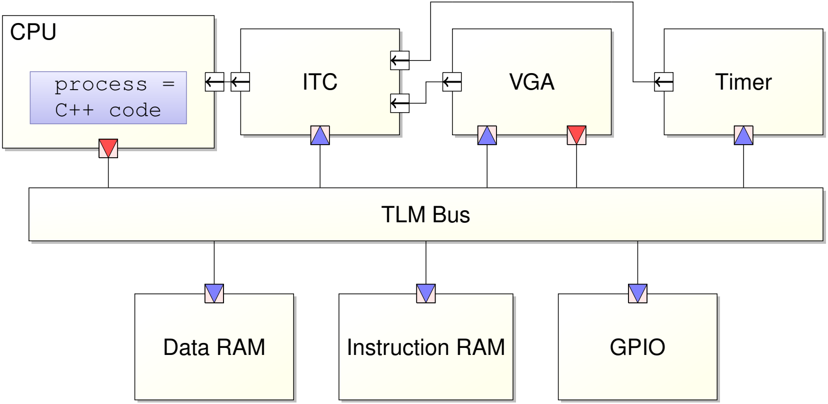
A first definition

- Model what is needed for Software Execution:
  - Processors
  - Address-map
  - Concurrency
- ... and only that.
  - No micro-architecture
  - No bus protocol
  - No pipeline
  - No physical clock
  - …

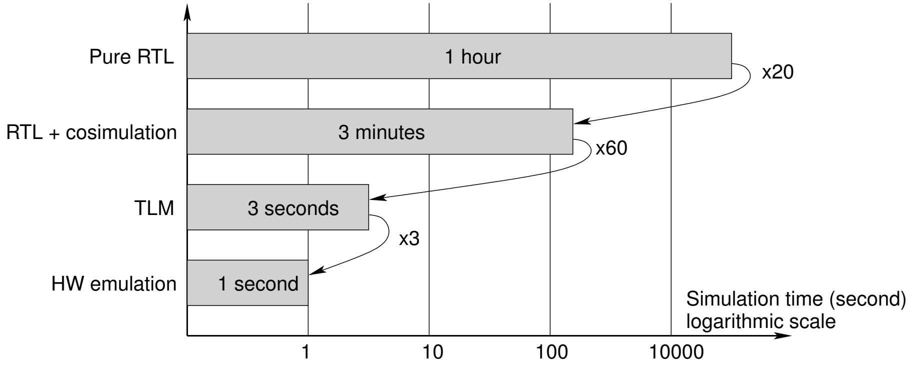




An example TLM Model



Performance of TLM



Uses of Functional Models

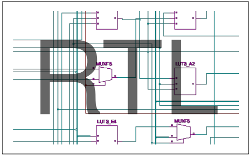
Reference for Hardware Validation

Virtual Prototype for Software Development

Uses of Functional Models

Reference for Hardware Validation

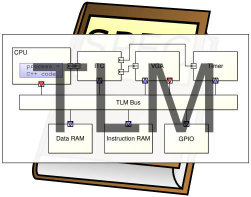
$\stackrel{?}{=}$



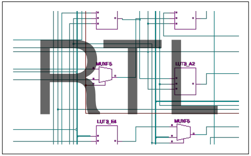
Virtual Prototype for Software Development

Uses of Functional Models

Reference for Hardware Validation



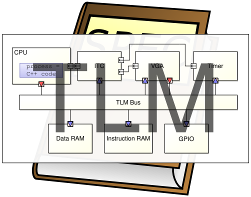
$\stackrel{?}{=}$



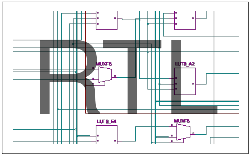
Virtual Prototype for Software Development

Uses of Functional Models

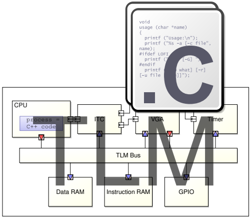
Reference for Hardware Validation



$\stackrel{?}{=}$

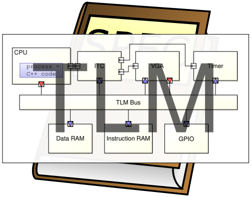


Virtual Prototype for Software Development

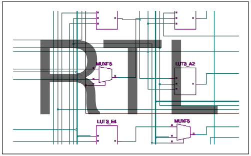


Uses of Functional Models

Reference for Hardware Validation

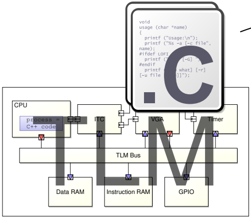


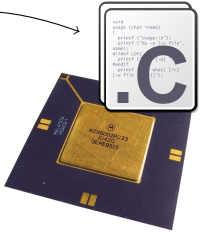
$\stackrel{?}{=}$



Unmodified Software

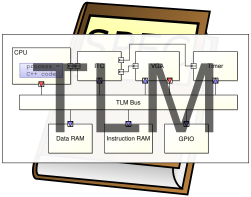
Virtual Prototype for Software Development



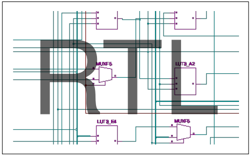


Uses of Functional Models

Reference for Hardware Validation

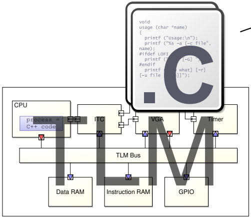


$\stackrel{?}{\equiv}$

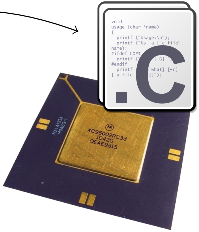


Virtual Prototype for Software Development

Unmodified Software



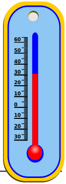
$\stackrel{?}{\supseteq}$



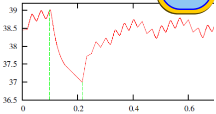
Content of a TLM Model

A richer definition

- Timing information
  - May be needed for Software Execution
  - Useful for Profiling Software

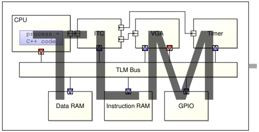


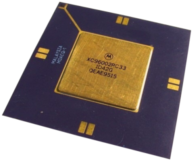
- Power and Temperature
  - Validate design choices
  - Validate power-management policy



Use of Non-Functional Models

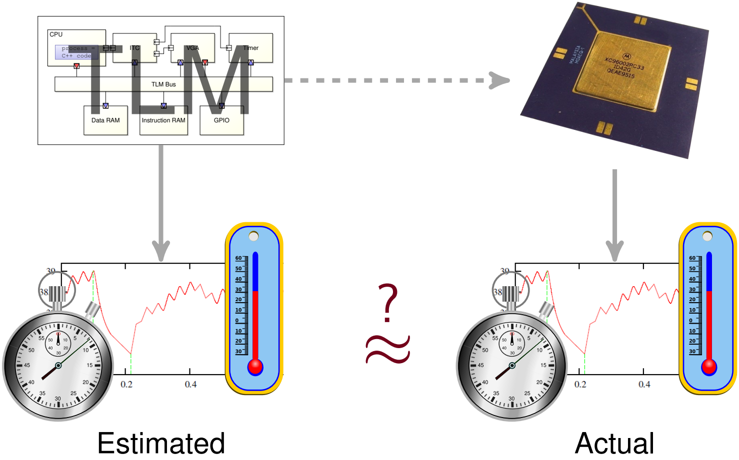
Timing, Power consumption, Temperature Estimation





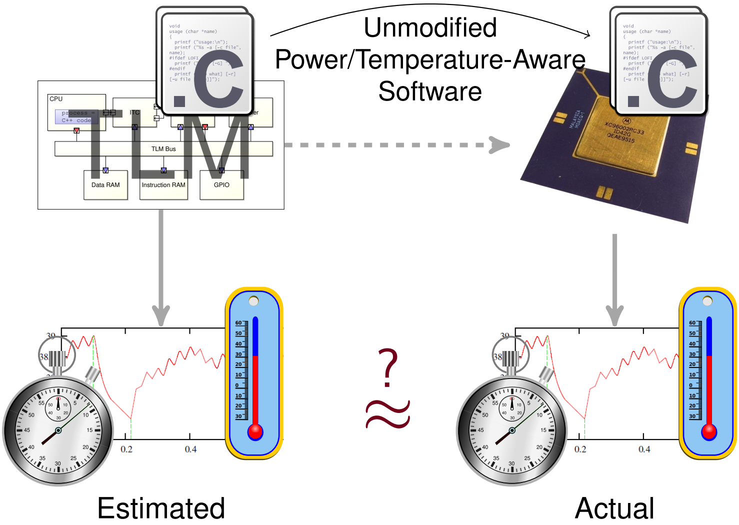
Use of Non-Functional Models

Timing, Power consumption, Temperature Estimation



Use of Non-Functional Models

Timing, Power consumption, Temperature Estimation



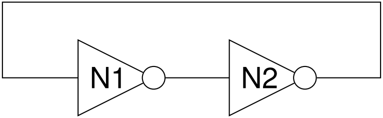
Summary: Expected Properties of TLM Programs

SystemC/TLM Programs should

- Simulate fast,
- Satisfy correctness criterions,
- Reflect faithfully functional and non-functional properties of the actual system.

Outline

SystemC: Simple Example




```
SC_MODULE(not_gate) {
    sc_in<bool> in;
    sc_out<bool> out;

    void compute (void) {
        // Behavior
        bool val = in.read();
        out.write(!val);
    }

    SC_CTOR(not_gate) {
        SC_METHOD(compute);
        sensitive << in;
    }
};
```

```
int sc_main(int argc, char **argv) {
    // Elaboration phase (Architecture)
    // Instantiate modules ...
    not_gate n1("N1");
    not_gate n2("N2");
    sc_signal<bool> s1, s2;
    // ... and bind them together
    n1.out.bind(s1);
    n2.out.bind(s2);
    n1.in.bind(s2);
    n2.in.bind(s1);

    // Start simulation
    sc_start(100, SC_NS);
    return 0;
}
```

Compiling SystemC

```
$ g++ example.cpp -lsystemc
$ ./a.out
```

... end of section?

Compiling SystemC

```
$ g++ example.cpp -lsystemc
$ ./a.out
```

But ...

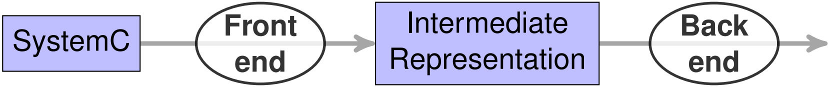
- C++ compilers cannot do SystemC-aware optimizations
- C++ analyzers do not know SystemC semantics

This section

2. Compilation of SystemC/TLM
   - Front-end
   - Optimization and Fast Simulation

SystemC Front-End

- In this talk: Front-end = “Compiler front-end” (AKA “Parser”)



Intermediate Representation = Architecture + Behavior

SystemC Front-Ends

- When you *don't* need a front-end:
  - Main application of SystemC: Simulation
  - Testing, run-time verification, monitoring...

SystemC Front-Ends

- When you *don't* need a front-end:
  - Main application of SystemC: Simulation
  - Testing, run-time verification, monitoring...

$\Rightarrow$ No reference front-end available on `http://systemc.org/`

SystemC Front-Ends

- When you *don't* need a front-end:
  - Main application of SystemC: Simulation
  - Testing, run-time verification, monitoring...

$\Rightarrow$ No reference front-end available on `http://systemc.org/`

- When you *do* need a front-end:
  - Symbolic formal verification, High-level synthesis
  - Visualization
  - Introspection
  - SystemC-specific Compiler Optimizations
  - Advanced debugging features

Challenges and Solutions with SystemC Front-Ends

1. C++ is complex (e.g. `clang` ≈ 200,000 LOC)

2. Architecture built at runtime, with C++ code

```
SC_MODULE(not_gate) {
    sc_in<bool> in;
    sc_out<bool> out;
    void compute (void) {
        // Behavior
        bool val = in.read();
        out.write(!val);
    }

    SC_CTOR(not_gate) {
        SC_METHOD(compute);
        sensitive << in;
    }
};
```

```
int sc_main(int argc, char **argv) {
    // Elaboration phase (Architecture)
    not_gate n1("N1");
    not_gate n2("N2");
    sc_signal<bool> s1, s2;
    // Binding
    n1.out.bind(s1);
    n2.out.bind(s2);
    n1.in.bind(s2);
    n2.in.bind(s1);

    // Start simulation
    sc_start(100, SC_NS); return 0;
}
```

Challenges and Solutions with SystemC Front-Ends

1. C++ is complex (e.g. `clang` ≈ 200,000 LOC)
   ⇝ Write a C++ front-end or reuse one (g++, clang, EDG, . . .)
2. Architecture built at runtime, with C++ code
   ⇝ Analyze elaboration phase or execute it

```
SC_MODULE(not_gate) {
    sc_in<bool> in;
    sc_out<bool> out;
    void compute (void) {
        // Behavior
        bool val = in.read();
        out.write(!val);
    }

    SC_CTOR(not_gate) {
        SC_METHOD(compute);
        sensitive << in;
    }
};
```

```
int sc_main(int argc, char **argv) {
    // Elaboration phase (Architecture)
    not_gate n1("N1");
    not_gate n2("N2");
    sc_signal<bool> s1, s2;
    // Binding
    n1.out.bind(s1);
    n2.out.bind(s2);
    n1.in.bind(s2);
    n2.in.bind(s1);

    // Start simulation
    sc_start(100, SC_NS); return 0;
}
```

Challenges and Solutions with SystemC Front-Ends

1. C++ is complex (e.g. `clang` $\approx$ 200,000 LOC)
   $\leadsto$ Write a C++ front-end or reuse one (g++, clang, EDG, . . .)
2. Architecture built at runtime, with C++ code
   $\leadsto$ Analyze elaboration phase or execute it

Static Approaches

Dynamic Approaches

```
SC_MODULE(no _gate) {
    sc_in<bo l> in;
    sc_out<b ol> out;
    void co ute (void) {
        //   avior
                     in.read();
                            );
    }

    SC_CTOR(not_gate) {
        SC_METHOD(compute);
        sensitive << in;
    }
};
```

```
int sc_main(in   argc, char **argv) {
    // Elaborati    phase (Architecture)
    not_gate n1("
    not_g
    sc_signal<bool> s1, s2;
    // Binding
    n1.out.bind(s1);
    n2.out.bind(s2);
    n1.in.bind(s2);
    n2.in.bind(s1);

    // Start simulation
    sc_start(100, SC_NS); return 0;
}
```

Dealing with the architecture

When it becomes tricky...

```
int sc_main(int argc, char **argv) {
    int n = atoi(argv[1]);
    int m = atoi(argv[2]);
    Node array[n][m];
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            array[i][j]
                = new Node(...);
            ...
        }
    }

    sc_start(100, SC_NS);
    return 0;
}
```

Dealing with the architecture

When it becomes tricky...

- Static approach: cannot deal with such code
- Dynamic approach: can extract the architecture for individual instances of the system

```
int sc_main(int argc, char **argv) {
    int n = atoi(argv[1]);
    int m = atoi(argv[2]);
    Node array[n][m];
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            array[i][j]
                = new Node(...);
            ...
        }
    }

    sc_start(100, SC_NS);
    return 0;
}
```

Dealing with the architecture

When it becomes *very* tricky. . .

```
void compute(void) {
    for (int i = 0; i < n; i++) {
        ports[i].write(true);
    }
    ...
}
```

Dealing with the architecture

When it becomes *very* tricky...

- One can unroll the loop to let `i` become constant,
- Undecidable in the general case.

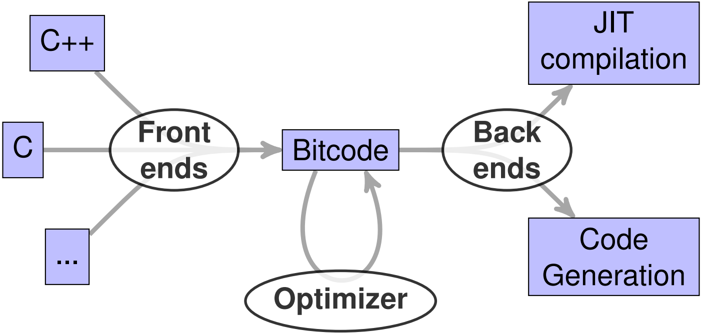
```
void compute(void) {
    for (int i = 0; i < n; i++) {
        ports[i].write(true);
    }
    ...
}
```

The beginning: Pinapa

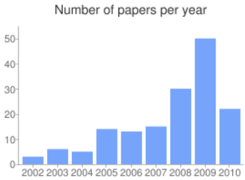
AKA "my Ph.D's front-end"

- Pinapa's principle:
  - Use GCC's C++ front-end
  - Compile, dynamically load and execute the elaboration (`sc_main`)
- Pinapa's drawbacks:
  - Uses GCC's internals (hard to port to newer versions)
  - Hard to install and use, no separate compilation
  - Ad-hoc match of SystemC constructs in AST
  - AST *Vs* SSA form in modern compilers

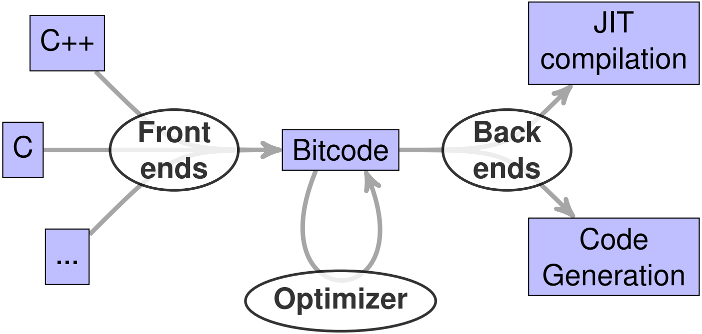
LLVM: Low Level Virtual Machine



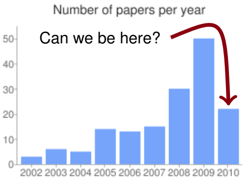
- Clean API
- Clean SSA intermediate representation
- Many tools available



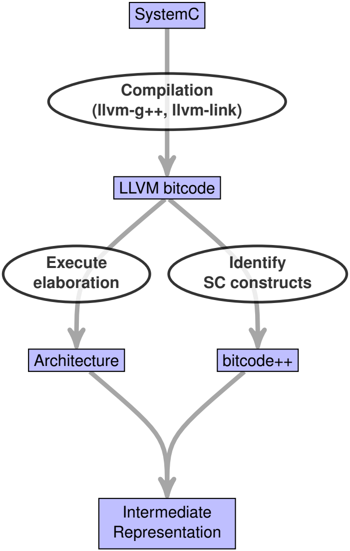
LLVM: Low Level Virtual Machine



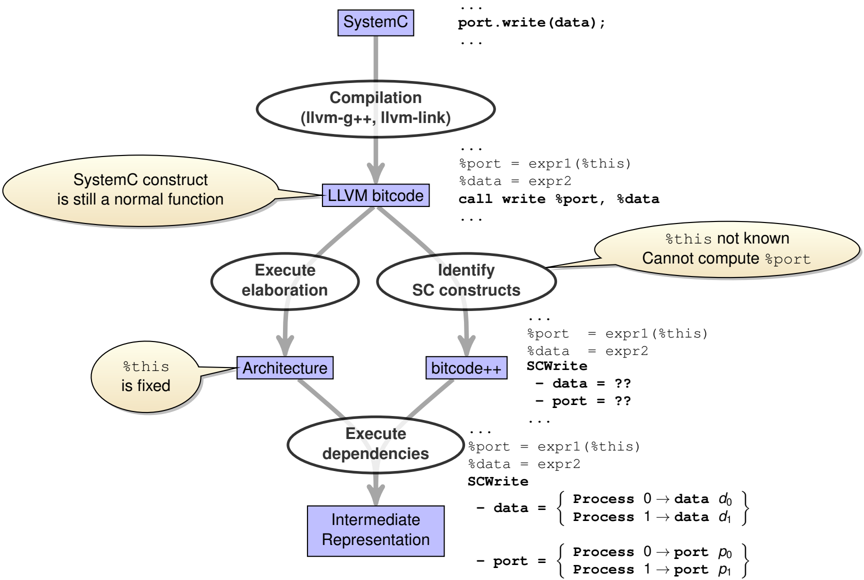
- Clean API
- Clean SSA intermediate representation
- Many tools available



PinaVM: Enriching the bitcode



PinaVM: Enriching the bitcode



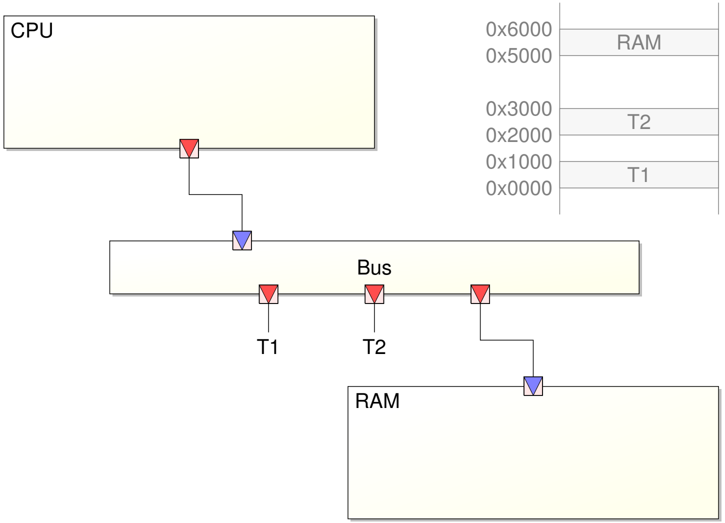
Summary

- PinaVM relies on executability (JIT Compiler) for execution of:
  - elaboration phase ($\approx$ like Pinapa)
  - sliced pieces of code
- Open Source: `http://forge.imag.fr/projects/pinavm/`
- Still a prototype, but very few fundamental limitations
- $\approx$ 3000 lines of C++ code on top of LLVM
- Experimental back-ends for
  - Execution (Tweto)
  - Model-checking (using SPIN)

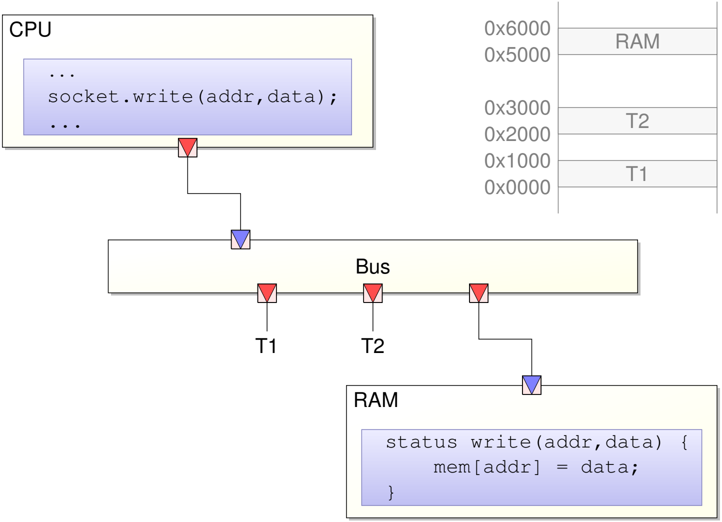
This section

2. Compilation of SystemC/TLM
   - Front-end
   - Optimization and Fast Simulation

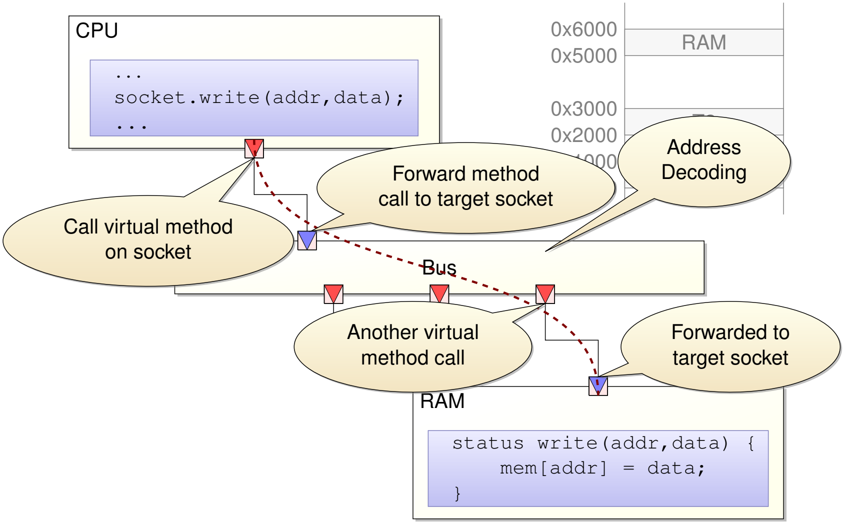
Typical Transaction Journey



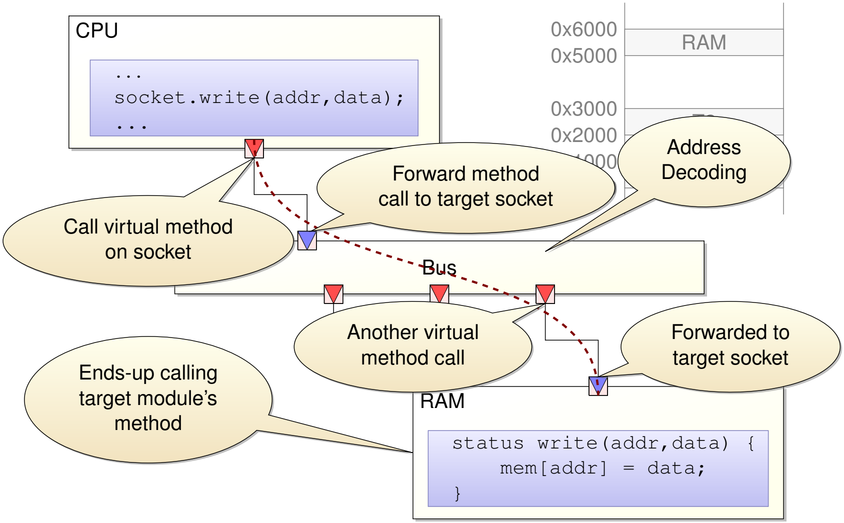
Typical Transaction Journey



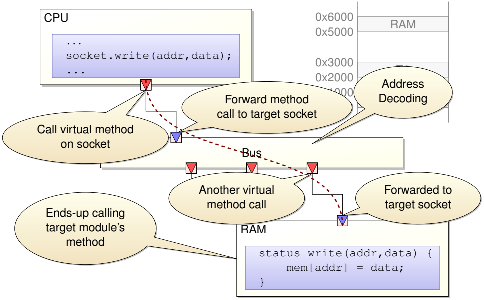
Typical Transaction Journey



Typical Transaction Journey

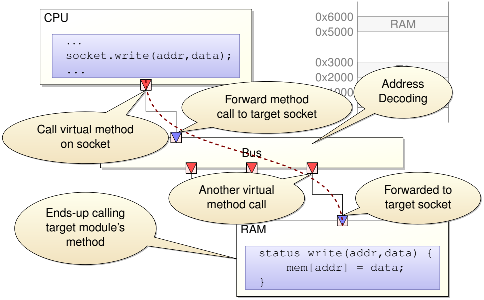


Typical Transaction Journey



- Many costly operations for a simple functionality
- Work-around: backdoor access (DMI = Direct Memory Interface)
  - CPU get a pointer to RAM's internal data
  - Manual, dangerous optimization

Typical Transaction Journey



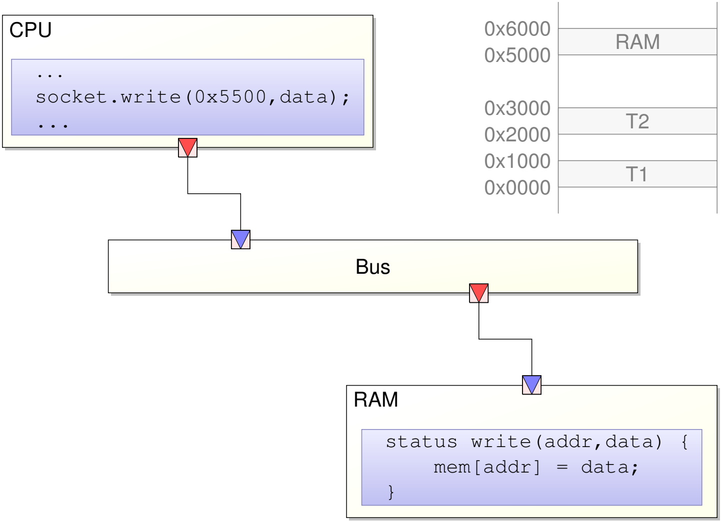
- Many costly operations for a simple functionality
- Work-around: backdoor access (DMI = Direct Memory Interface)
  - CPU get a pointer to RAM's internal data
  - Manual, dangerous optimization

Can a compiler be as good as DMI,
automatically and safely?

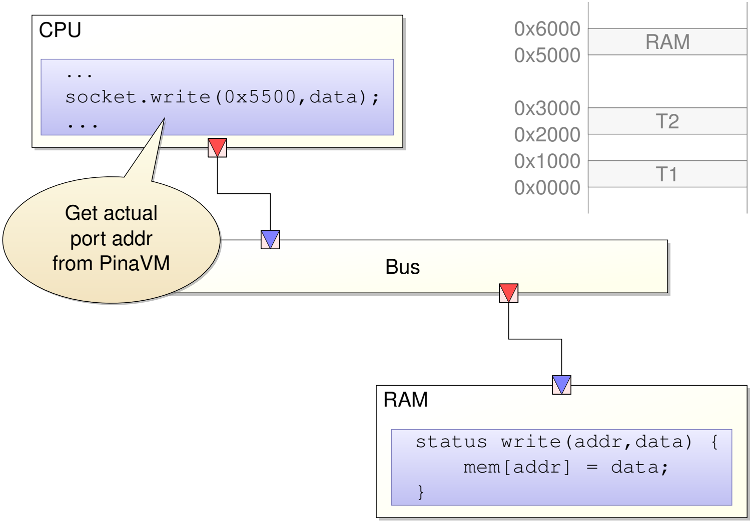
Basic Ideas

- Do statically what can be done statically ...
- ... considering “statically” = “after elaboration”
- Examples:
  - Virtual function resolution
  - Inlining through SystemC ports
  - Static address resolution

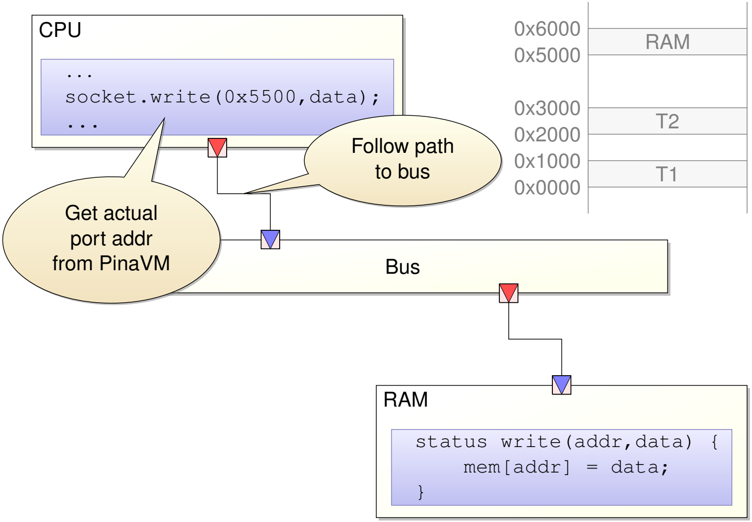
Dealing with addresses *Statically*



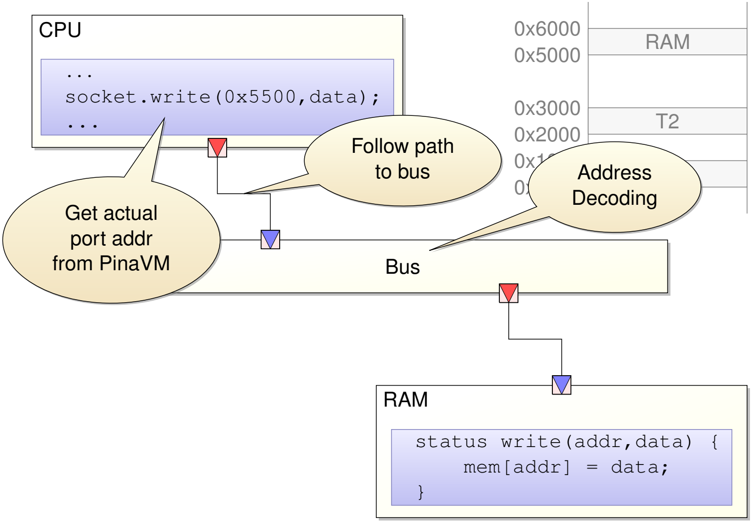
Dealing with addresses *Statically*



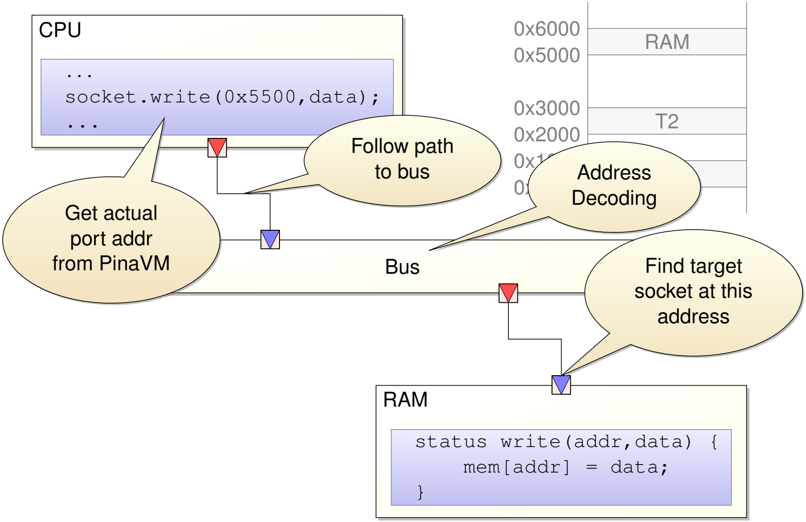
Dealing with addresses *Statically*



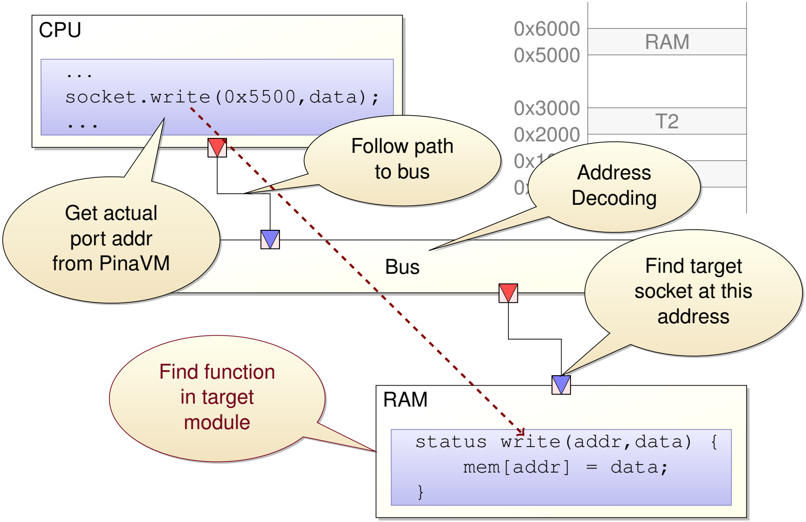
Dealing with addresses *Statically*



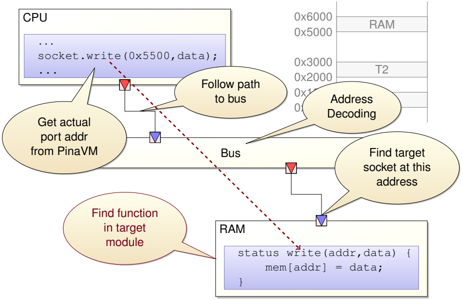
Dealing with addresses *Statically*



Dealing with addresses *Statically*

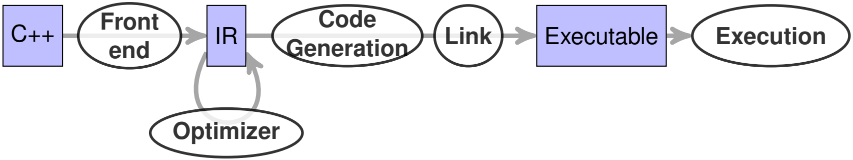


Dealing with addresses *Statically*

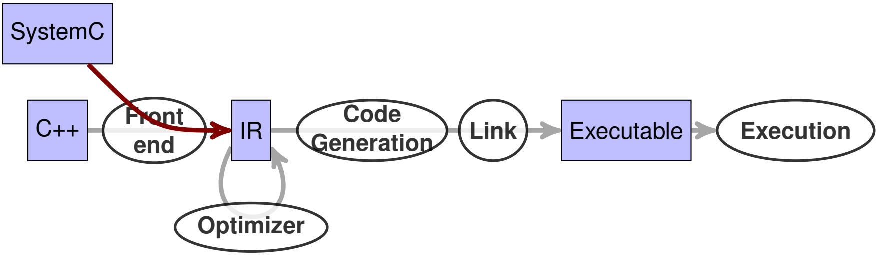


- Possible optimizations:
  - Replace call to `socket.write()` with `RAM.write()`
  - Possibly inline it

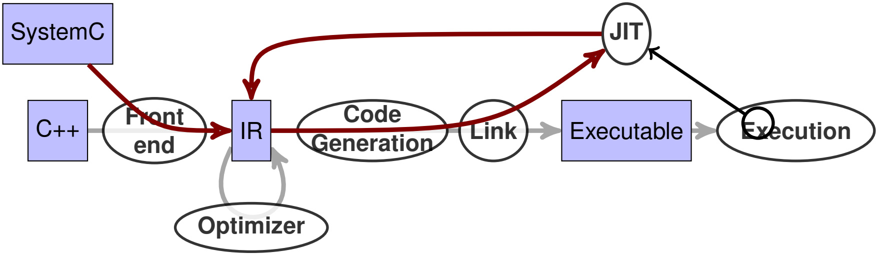
Optimized Compilation for SystemC



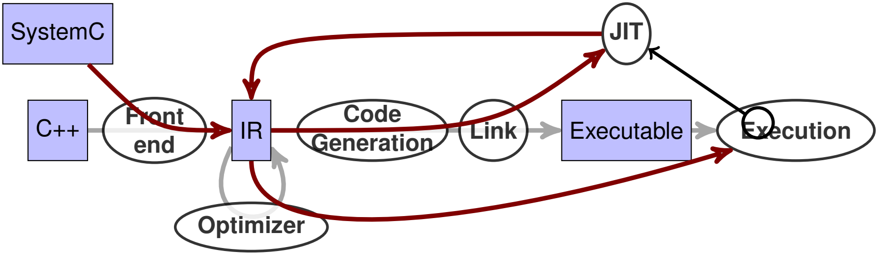
Optimized Compilation for SystemC



Optimized Compilation for SystemC

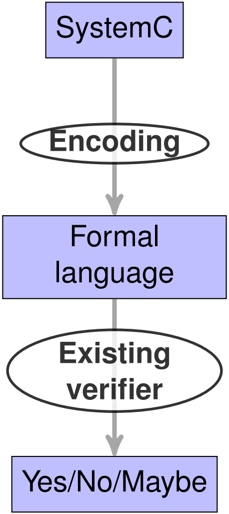


Optimized Compilation for SystemC

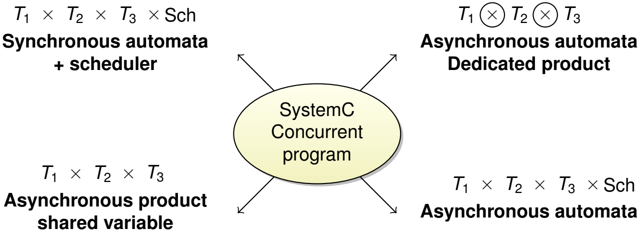


Outline

Encoding Approaches



Encoding Approaches



Encoding Approaches

$T_1 \times T_2 \times T_3 \times \mathrm{Sch}$
Synchronous automata + scheduler

$T_1 \otimes T_2 \otimes T_3$
Asynchronous automata Dedicated product

SystemC Concurrent program

$T_1 \times T_2 \times T_3$
Asynchronous product shared variable

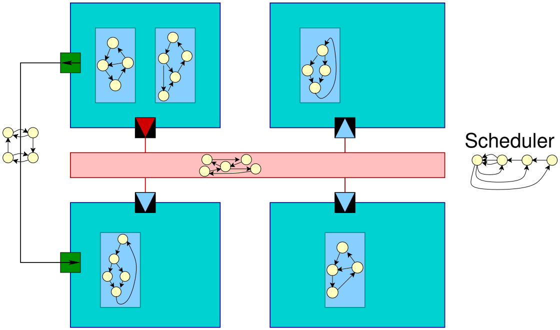
$T_1 \times T_2 \times T_3 \times \mathrm{Sch}$
Asynchronous automata

Translating a SystemC Program

- Translation = Parse the source code, generate an automaton
- Direct semantics = Read the specification, instantiate an automaton

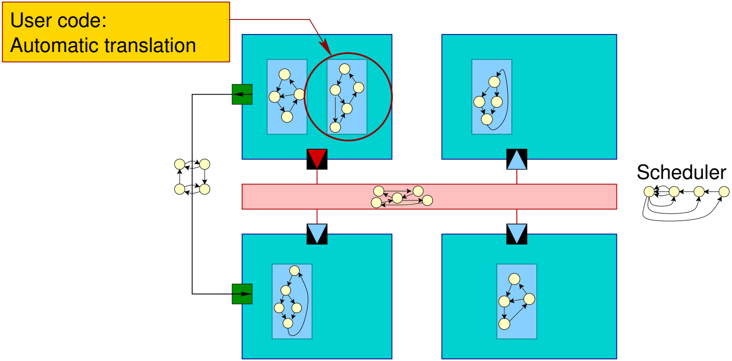
Translating a SystemC Program

- Translation = Parse the source code, generate an automaton
- Direct semantics = Read the specification, instantiate an automaton



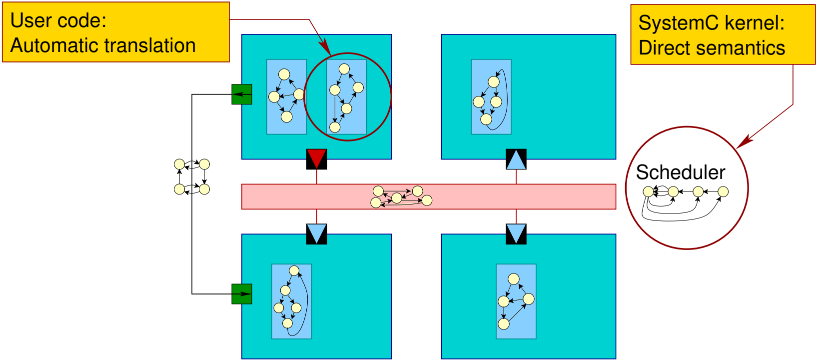
Translating a SystemC Program

- Translation = Parse the source code, generate an automaton
- Direct semantics = Read the specification, instantiate an automaton



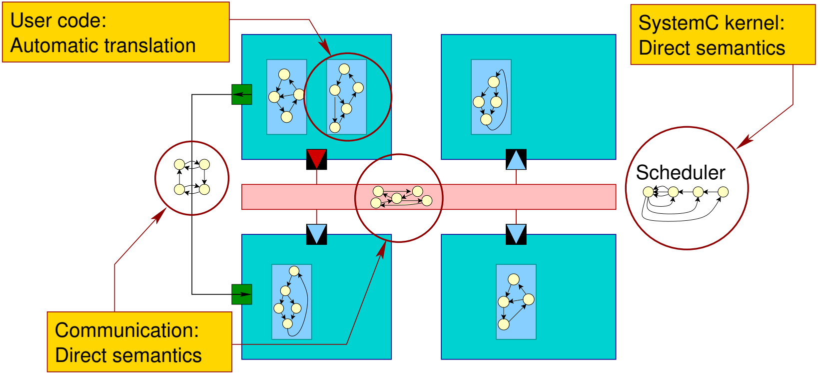
Translating a SystemC Program

- Translation = Parse the source code, generate an automaton
- Direct semantics = Read the specification, instantiate an automaton



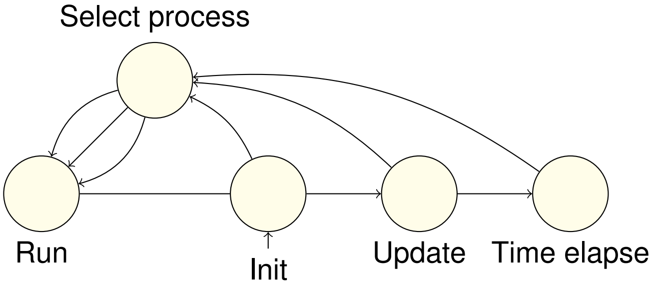
Translating a SystemC Program

- Translation = Parse the source code, generate an automaton
- Direct semantics = Read the specification, instantiate an automaton



The SystemC scheduler

- Non-preemptive scheduler
- Non-deterministic processes election



(+ 1 automaton per process to reflect its state)

Encoding Approaches

$T_1 \times T_2 \times T_3 \times \text{Sch}$
Synchronous automata + scheduler

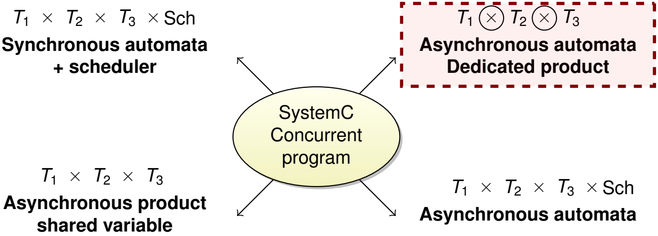
$T_1 \otimes T_2 \otimes T_3$
Asynchronous automata Dedicated product

SystemC Concurrent program

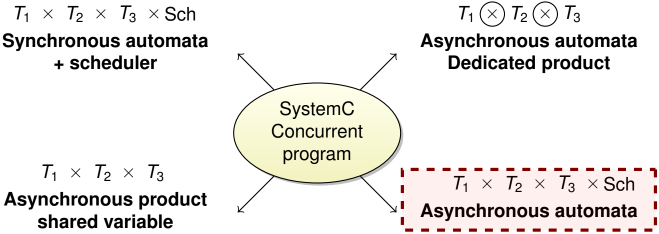
$T_1 \times T_2 \times T_3$
Asynchronous product shared variable

$T_1 \times T_2 \times T_3 \times \text{Sch}$
Asynchronous automata

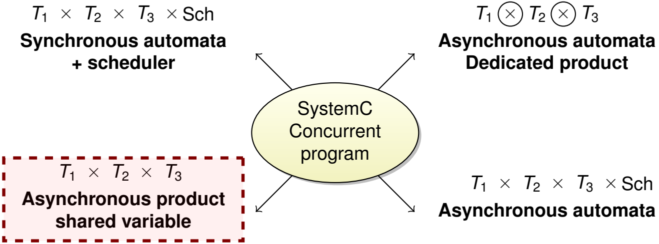
Encoding Approaches



Encoding Approaches



Encoding Approaches



SystemC to Spin: encoding events

- notify/wait for event E^k:

p**::wait(**E^k**):** $W_p := k$ $\text{blocked}(W_p == 0)$	p**::notify(**E^k**):** $\forall i \in P \vert W_i == K$ $W_i := 0$

- W_p : integer associated to process p.
  $W_p = k \Leftrightarrow$ "process p is waiting for event E^k".

SystemC to Spin: encoding time and events

- discrete time
- a deadline variable T_p is attached to each process p
  T_p = next execution time for process p

p::**wait(d):**
$T_p := T_p + d$
blocked($T_p == \min_{i \in P}(T_i)$)

"Set my next execution time to now + d and wait until the current execution time reaches it"

SystemC to Spin: encoding time and events

- discrete time
- a deadline variable T_p is attached to each process p
  T_p = next execution time for process p

p::wait(d):

$$T_p := T_p + d$$
$$\text{blocked}(T_p == \min_{\substack{i \in P \\ \mathbf{W_i == 0}}} (T_i))$$

"Set my next execution time to now $+ d$ and wait until the current execution time reaches it"

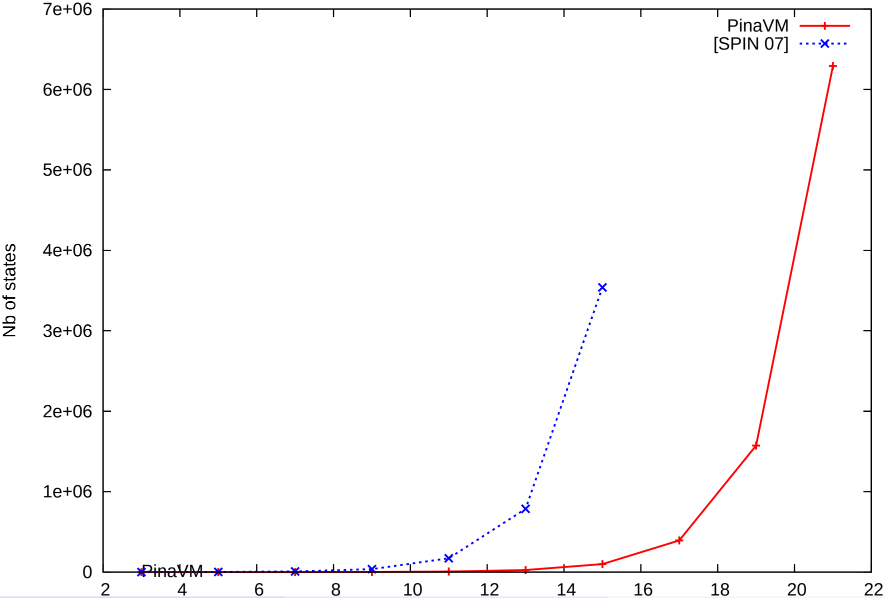
p::wait(E^k):

$$W_p := \text{K}$$
$$\text{blocked}(W_p == 0)$$

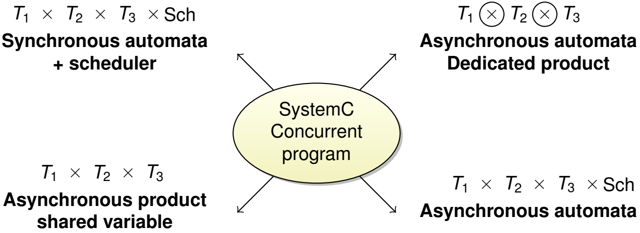
p::notify(E^k):

$$\forall i \in P | W_i == K$$
$$W_i := 0$$
$$T_i := T_p$$

SystemC to Spin: results



Encoding Approaches



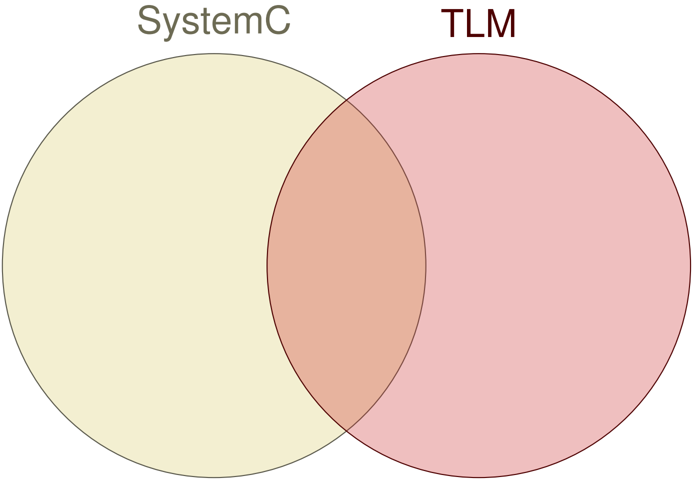
Outline

This section

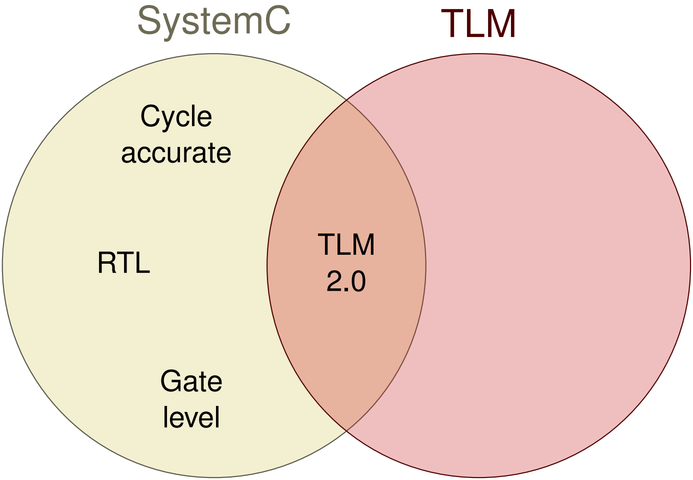
4 Non-functional Properties in TLM

- Time and Concurrency
  - jTLM
  - Parallelization: jTLM and SC-DURING
- Power and Temperature Estimation

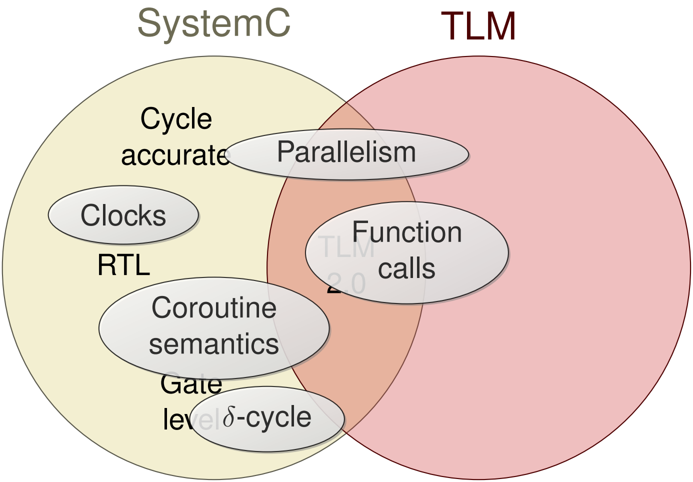
SystemC/TLM vs. “TLM Abstraction Level”



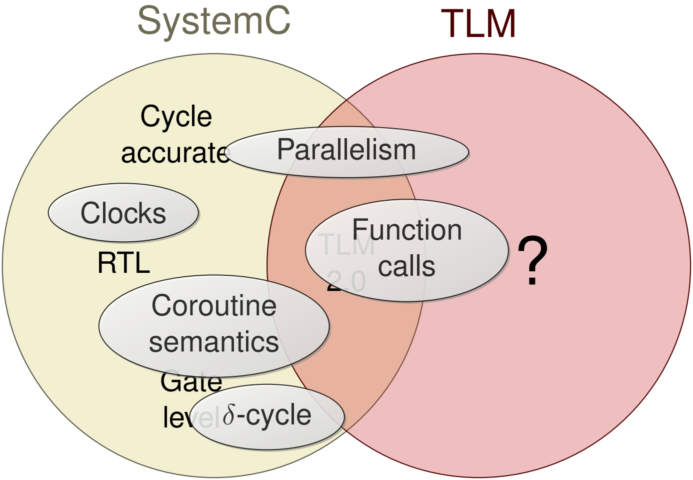
SystemC/TLM vs. "TLM Abstraction Level"



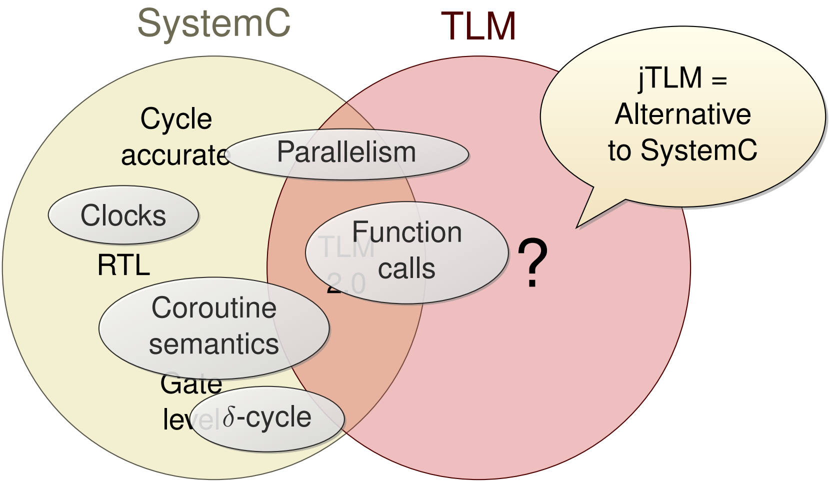
SystemC/TLM vs. "TLM Abstraction Level"



SystemC/TLM vs. "TLM Abstraction Level"



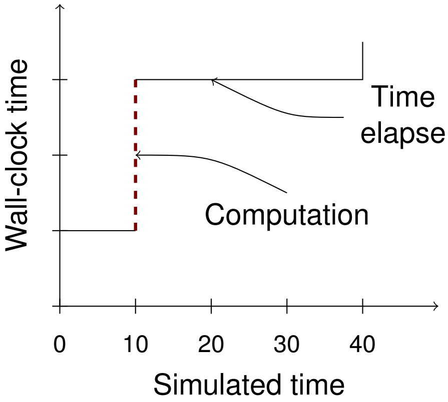
SystemC/TLM vs. "TLM Abstraction Level"



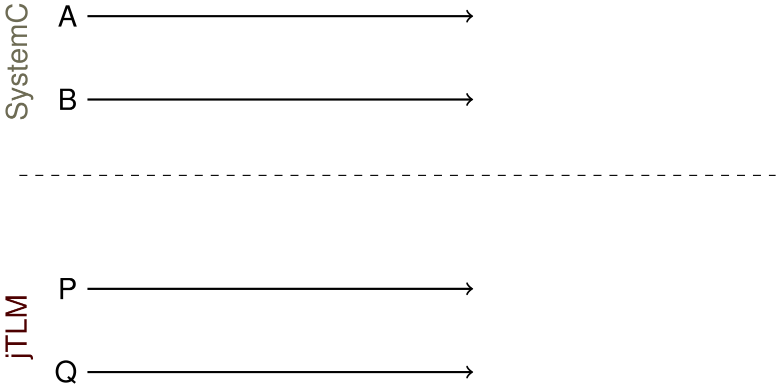
jTLM: goals and peculiarities

- jTLM's initial goal: define "TLM" independently of SystemC
  - Not cooperative (true parallelism)
  - Not C++ (Java)
  - No δ-cycle
- Interesting features
  - Small and simple code ($\approx$ 500 LOC)
  - Nice experimentation platform
- Not meant for production

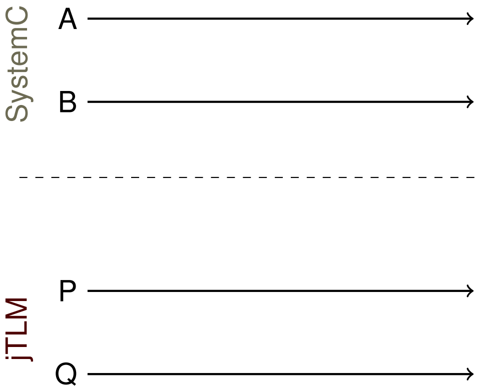
Simulated Time Vs Wall-Clock Time



(Simulated) Time in SystemC and jTLM



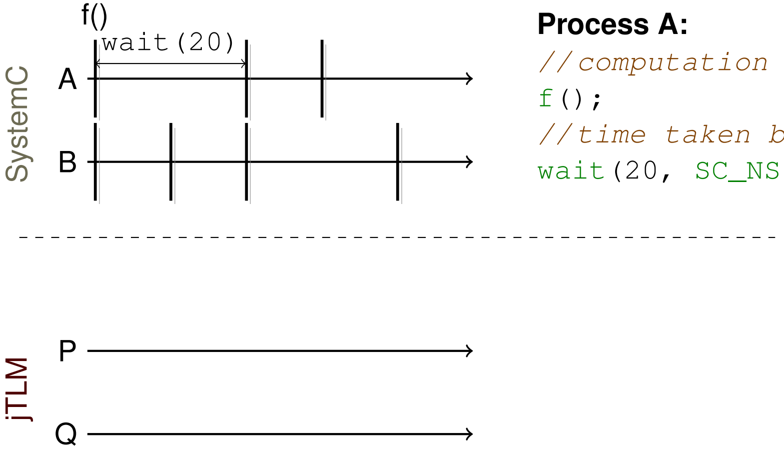
(Simulated) Time in SystemC and jTLM



Process A:

```
//computation
f();
//time taken by f
wait(20, SC_NS);
```

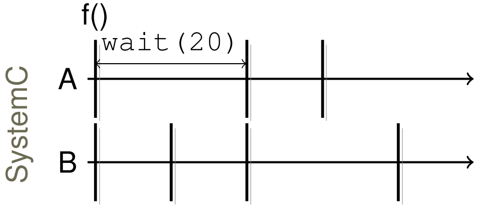
(Simulated) Time in SystemC and jTLM



Process A:

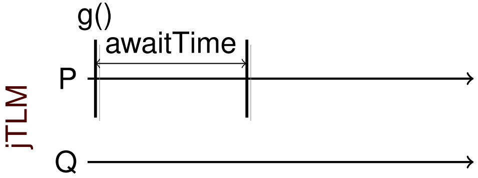
```
//computation
f();
//time taken by f
wait(20, SC_NS);
```

(Simulated) Time in SystemC and jTLM



Process A:

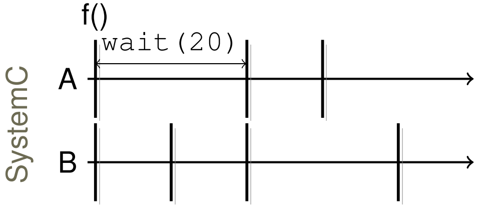
```
//computation
f();
//time taken by f
wait(20, SC_NS);
```



Process P:

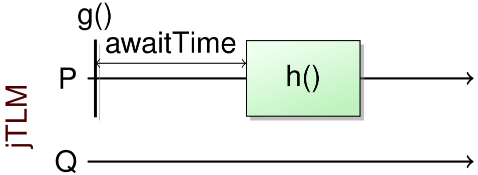
```
g();
awaitTime(20);
```

(Simulated) Time in SystemC and jTLM



Process A:

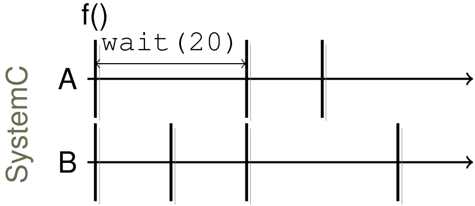
```
// computation
f();
// time taken by f
wait(20, SC_NS);
```



Process P:

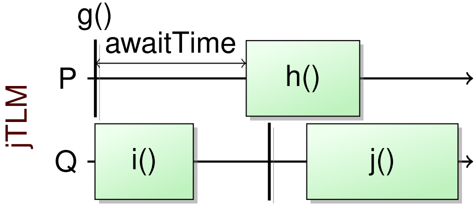
```
g();
awaitTime(20);
consumesTime(15) {
  h();
}
```

(Simulated) Time in SystemC and jTLM



Process A:

```
//computation
f();
//time taken by f
wait(20, SC_NS);
```

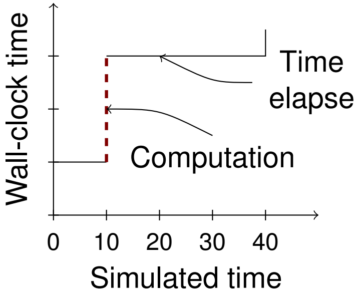


Process P:

```
g();
awaitTime(20);
consumesTime(15) {
  h();
}
```

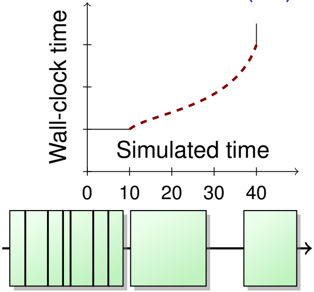
Time *à la* SystemC: `awaitTime(T)`

- By default, time does not elapse ⇒ instantaneous tasks
- `awaitTime(T)` : suspend and let other processes execute for T time units




```
f(); // instantaneous
awaitTime(20);
```

Task with Known Duration: `consumesTime(T)`



- Semantics:
  - Start and end dates known
  - Actions contained in task spread in between
- Advantages:
  - Model closer to actual system
  - Less bugs hidden
  - Better parallelization

```
consumesTime(15) {
    f1();
    f2();
    f3();
}
consumesTime(10) {
    g();
}
```

Addressing the Faithfulness Issue: Exposing Bugs

Example bug: mis-placed synchronization:

```
imgReady = true;          while(!imgReady)
awaitTime(5);          ||     awaitTime(1);
writeIMG();               awaitTime(10);
awaitTime(10);            readIMG();
```

⇒ bug never seen in simulation

Addressing the Faithfulness Issue: Exposing Bugs

Example bug: mis-placed synchronization:

```
imgReady = true;           while(!imgReady)
awaitTime(5);          ||      awaitTime(1);
writeIMG();                awaitTime(10);
awaitTime(10);             readIMG();
```

⇒ bug never seen in simulation

```
consumesTime(15) {            while(!imgReady)
    imgReady = true;              awaitTime(1);
    writeIMG();           ||  awaitTime(10);
}                             readIMG();
```

⇒ strictly more behaviors, including the buggy one

Time Queue and `awaitTime(T)`

Process P:

```
►f();
 awaitTime(50);
```

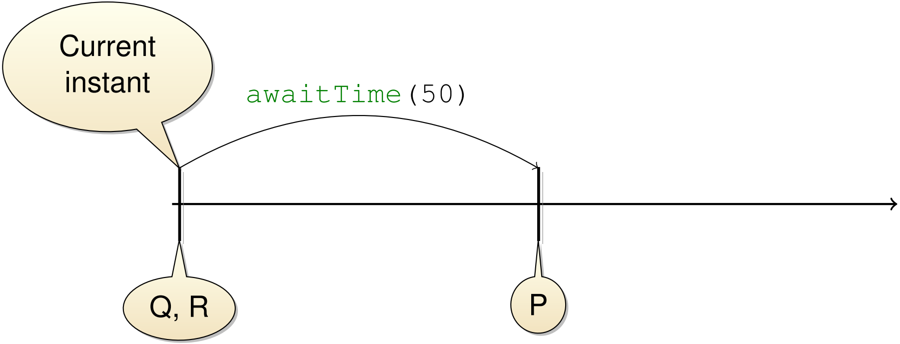
Process Q:

```
►h();
 awaitTime(30);
 g();
 awaitTime(30);
```

Process R:

```
►i();
 awaitTime(90);
```

Time Queue and `awaitTime(T)`

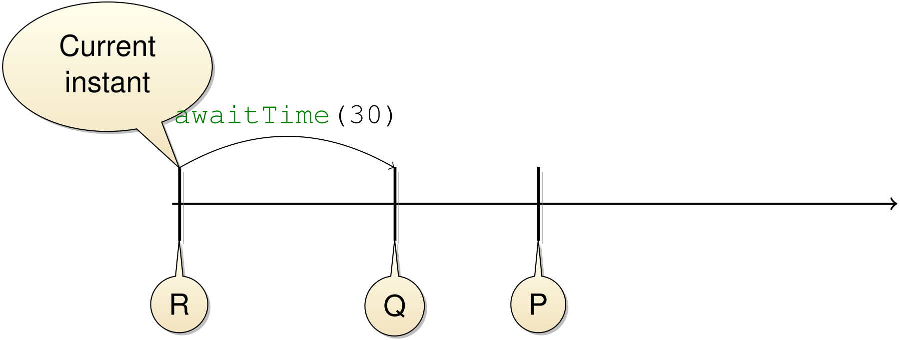



```
Process P:
  f();
▷awaitTime(50);
```

```
Process Q:
►h();
 awaitTime(30);
 g();
 awaitTime(30);
```

```
Process R:
►i();
 awaitTime(90);
```

Time Queue and `awaitTime(T)`

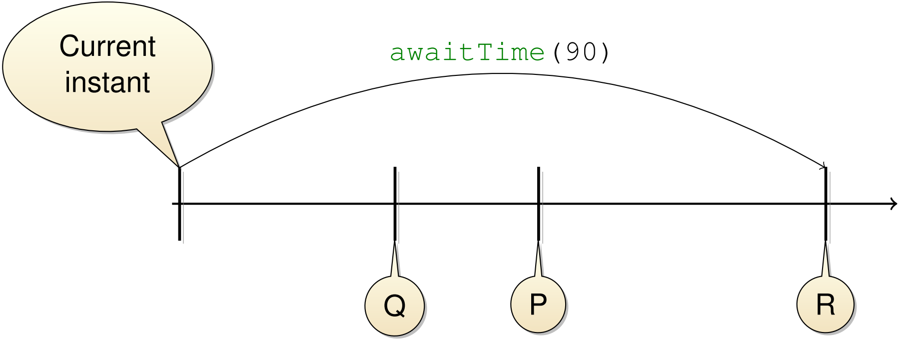


Process P:
```
 f();
▷awaitTime(50);
```

Process Q:
```
 h();
▷awaitTime(30);
 g();
 awaitTime(30);
```

Process R:
```
▶i();
 awaitTime(90);
```

Time Queue and `awaitTime(T)`

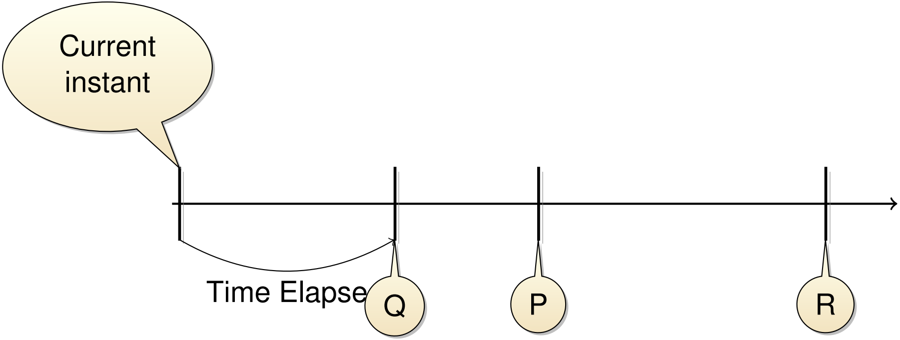


Process P:
```
  f();
▷awaitTime(50);
```

Process Q:
```
  h();
▷awaitTime(30);
  g();
  awaitTime(30);
```

Process R:
```
  i();
▷awaitTime(90);
```

Time Queue and `awaitTime(T)`

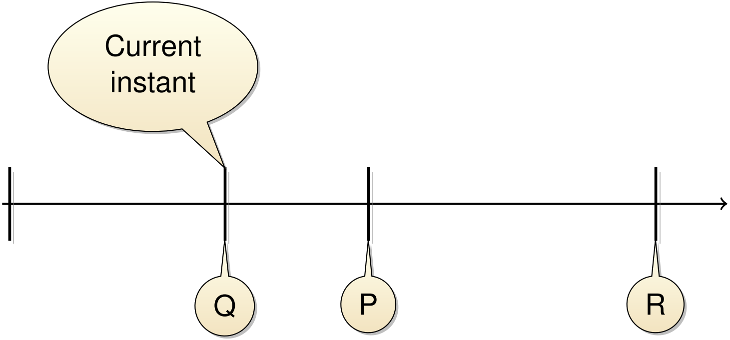


Process P:
```
 f();
▷awaitTime(50);
```

Process Q:
```
 h();
▶awaitTime(30);
 g();
 awaitTime(30);
```

Process R:
```
 i();
▷awaitTime(90);
```

Time Queue and `awaitTime(T)`



Process P:
```
  f();
▷awaitTime(50);
```

Process Q:
```
  h();
  awaitTime(30);
►g();
  awaitTime(30);
```

Process R:
```
  i();
▷awaitTime(90);
```

Time Queue and `consumesTime(T)`

What about `consumesTime(T)`?

Time Queue and `consumesTime(T)`

Process P:

```
▶f();
 consumesTime(50){
   g();
 }
 h();
```

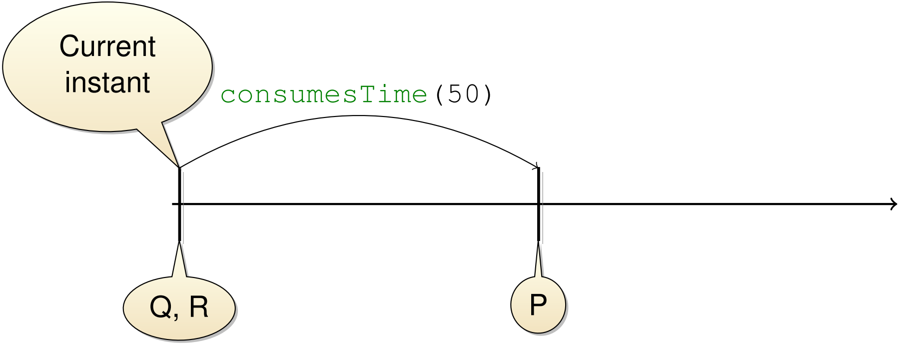
Process Q:

```
▶i();
 awaitTime(30);
 j();
 consumesTime(30){
   k();
 }
```

Process R:

```
▶l();
 awaitTime(90);
```

Time Queue and `consumesTime(T)`

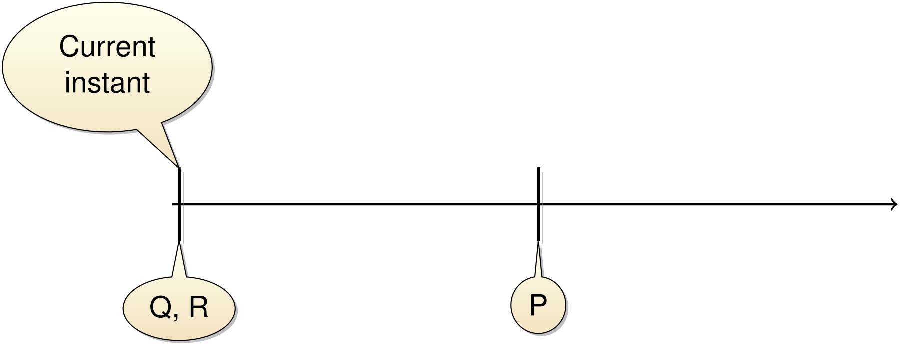


Process P:
```
  f();
► consumesTime(50){
    g();
  }
  h();
```

Process Q:
```
► i();
  awaitTime(30);
  j();
  consumesTime(30){
    k();
  }
```

Process R:
```
► l();
  awaitTime(90);
```

Time Queue and `consumesTime(T)`



Process P:

```
  f();
  consumesTime(50){
►   g();
  }
  h();
```

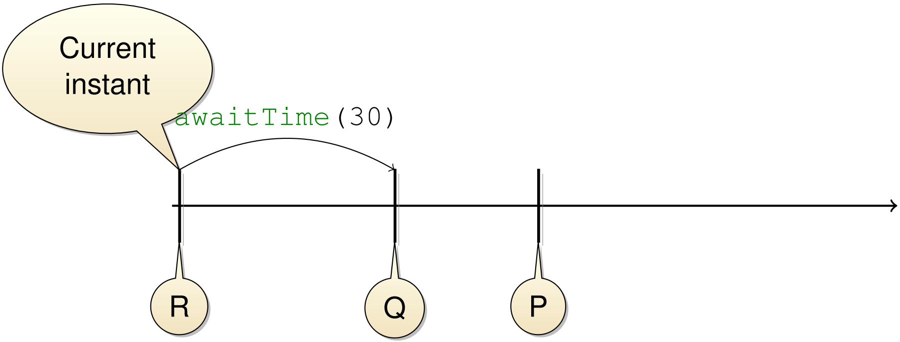
Process Q:

```
►i();
 awaitTime(30);
 j();
 consumesTime(30){
   k();
 }
```

Process R:

```
►l();
 awaitTime(90);
```

Time Queue and `consumesTime(T)`

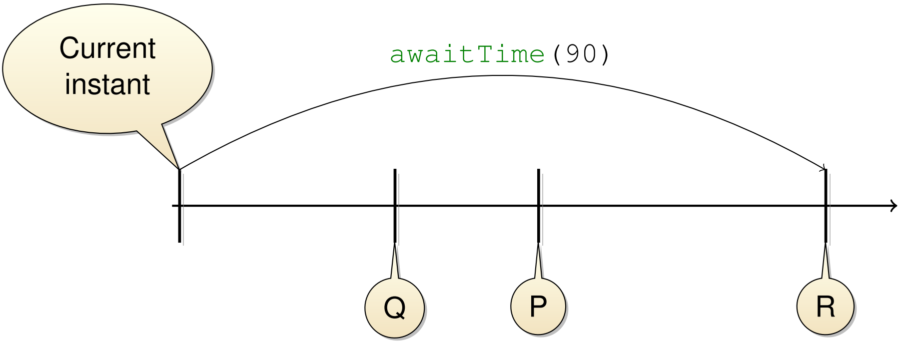


Process P:
```
  f();
  consumesTime(50){
►   g();
  }
  h();
```

Process Q:
```
  i();
▷ awaitTime(30);
  j();
  consumesTime(30){
    k();
  }
```

Process R:
```
► l();
  awaitTime(90);
```

Time Queue and `consumesTime(T)`

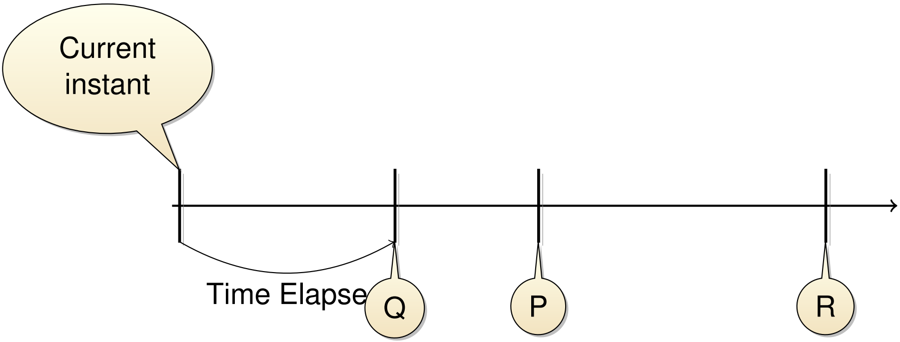



```
Process P:
  f();
  consumesTime(50){
▶   g();
  }
  h();
```

```
Process Q:
  i();
▷awaitTime(30);
  j();
  consumesTime(30){
    k();
  }
```

```
Process R:
  l();
▷awaitTime(90);
```

Time Queue and `consumesTime(T)`

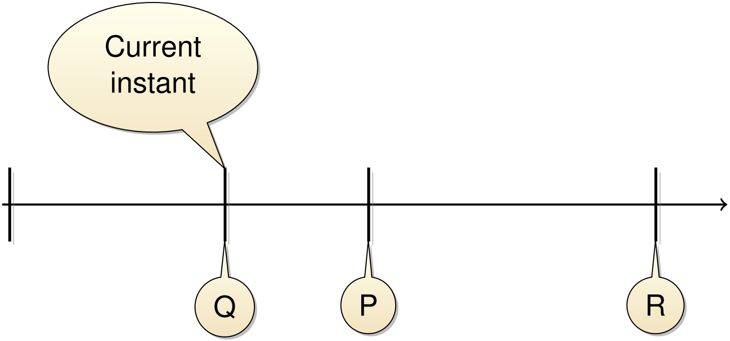



```
Process P:
  f();
  consumesTime(50){
►   g();
  }
  h();
```

```
Process Q:
  i();
►awaitTime(30);
  j();
  consumesTime(30){
    k();
  }
```

```
Process R:
  l();
▷awaitTime(90);
```

Time Queue and `consumesTime(T)`

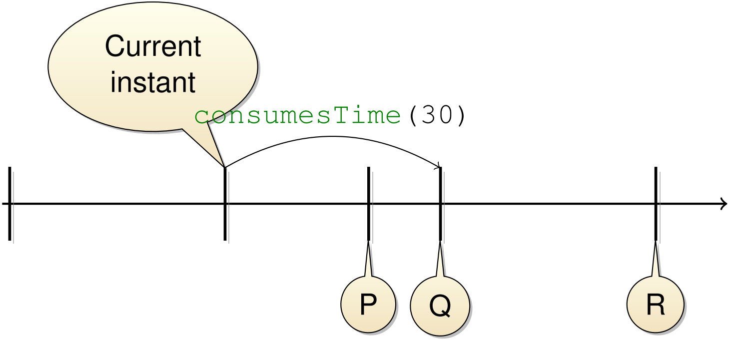



```
Process P:
  f();
  consumesTime(50){
►   g();
  }
  h();
```

```
Process Q:
  i();
  awaitTime(30);
► j();
  consumesTime(30){
    k();
  }
```

```
Process R:
  l();
▷ awaitTime(90);
```

Time Queue and `consumesTime(T)`

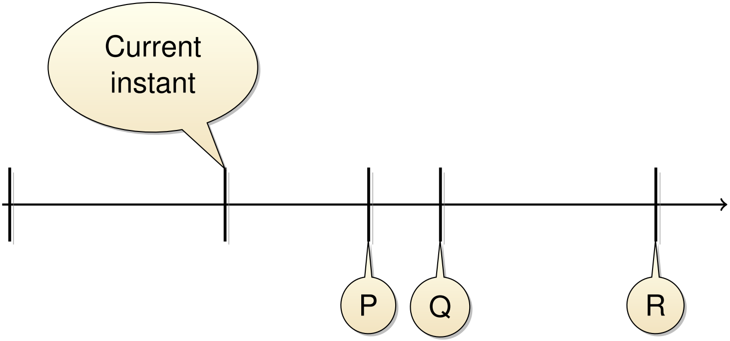


Process P:
```
  f();
  consumesTime(50){
►   g();
  }
  h();
```

Process Q:
```
  i();
  awaitTime(30);
  j();
► consumesTime(30){
    k();
  }
```

Process R:
```
  l();
▷ awaitTime(90);
```

Time Queue and `consumesTime(T)`

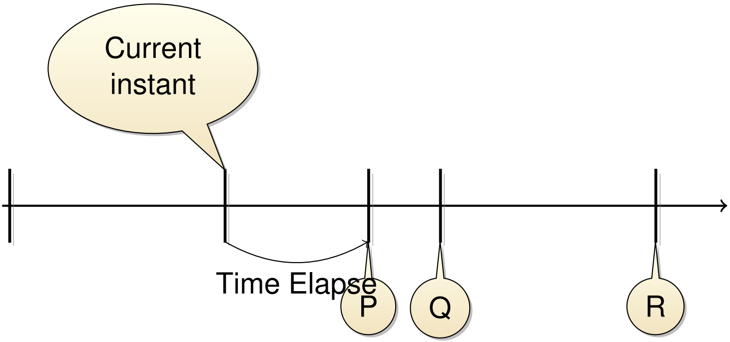



```
Process P:
  f();
  consumesTime(50){
►   g();
  }
  h();
```

```
Process Q:
  i();
  awaitTime(30);
  j();
  consumesTime(30){
►   k();
  }
```

```
Process R:
  l();
▷awaitTime(90);
```

Time Queue and `consumesTime(T)`

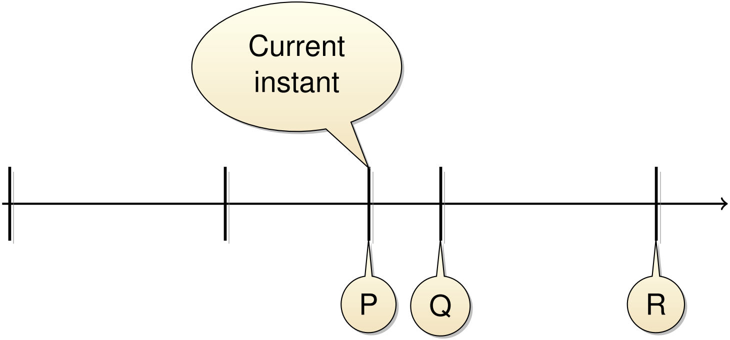



```
Process P:
  f();
  consumesTime(50){
    g();
► }
  h();
```

```
Process Q:
  i();
  awaitTime(30);
  j();
  consumesTime(30){
►   k();
  }
```

```
Process R:
  l();
▷awaitTime(90);
```

Time Queue and `consumesTime(T)`

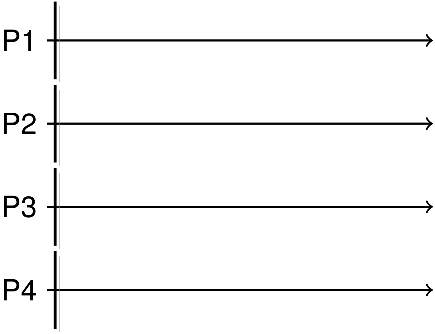


Process P:
```
 f();
 consumesTime(50){
   g();
 }
►h();
```

Process Q:
```
 i();
 awaitTime(30);
 j();
 consumesTime(30){
►  k();
 }
```

Process R:
```
 l();
▷awaitTime(90);
```

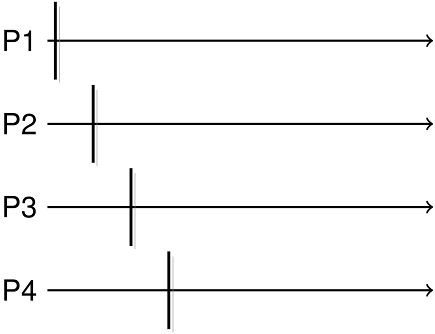
Parallelization



jTLM's Semantics

- Simultaneous tasks run in parallel

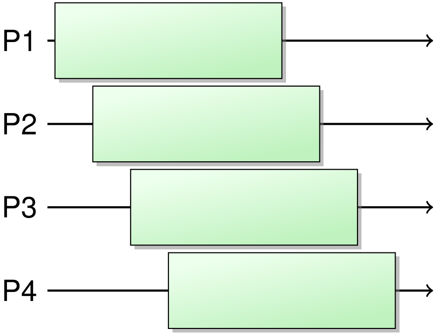
Parallelization



jTLM's Semantics

- Simultaneous tasks run in parallel
- Non-simultaneous tasks don't

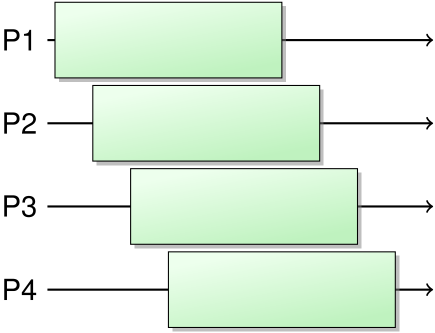
Parallelization



jTLM's Semantics

- Simultaneous tasks run in parallel
- Non-simultaneous tasks don't
- Overlapping tasks do

Parallelization

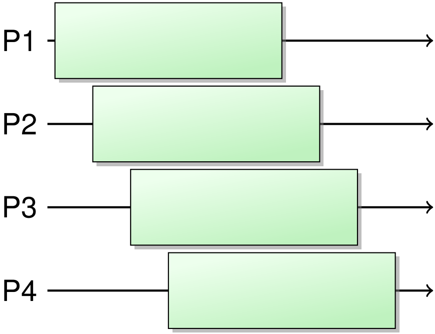


jTLM's Semantics

- Simultaneous tasks run in parallel
- Non-simultaneous tasks don't
- Overlapping tasks do

- Back to SystemC:
  - Parallelizing within δ-cycle = great if you have clocks
  - Simulated time is the bottleneck with quantitative/fuzzy time

Parallelization



jTLM's Semantics

- Simultaneous tasks run in parallel
- Non-simultaneous tasks don't
- Overlapping tasks do

- Back to SystemC:
  - Parallelizing within δ-cycle = great if you have clocks
  - Simulated time is the bottleneck with quantitative/fuzzy time

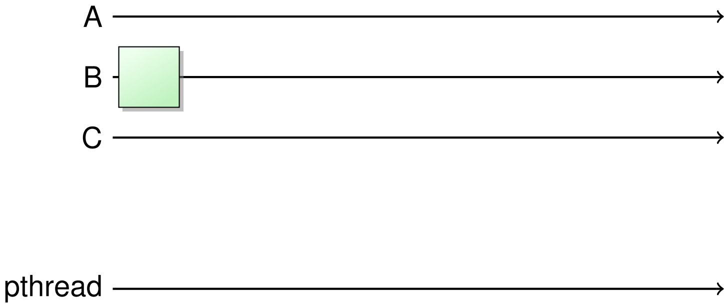
Can we apply the idea of duration to SystemC?

SC-DURING: the Idea

- Goal: allow during tasks in SystemC
  - Without modifying SystemC
  - Allowing physical parallelism
- Idea: let SystemC processes delegate computation to a separate thread

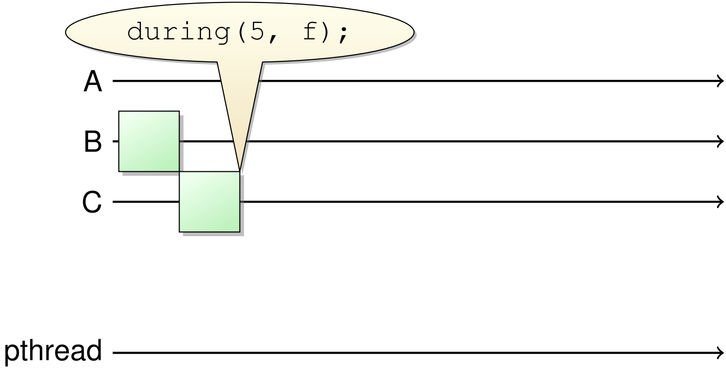
SC-DURING: Sketch of Implementation

```
void during(sc_core::sc_time duration,
        boost::function<void()> routine) {
①    boost::thread t(routine); // create thread
②    sc_core::wait(time); // let SystemC execute
③    t.join(); // wait for thread completion
}
```



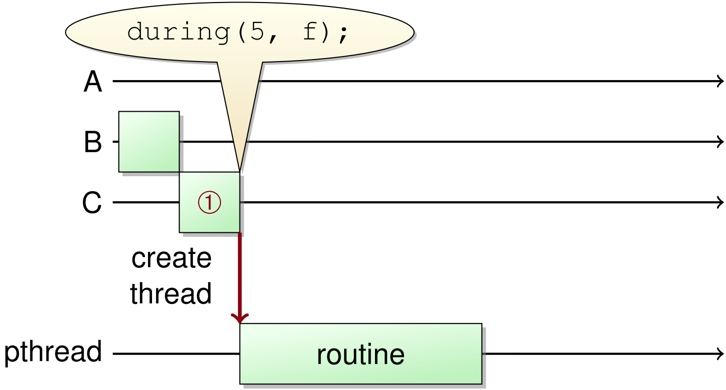
SC-DURING: Sketch of Implementation

```
    void during(sc_core::sc_time duration,
                boost::function<void()> routine) {
①       boost::thread t(routine); // create thread
②       sc_core::wait(time); // let SystemC execute
③       t.join(); // wait for thread completion
    }
```



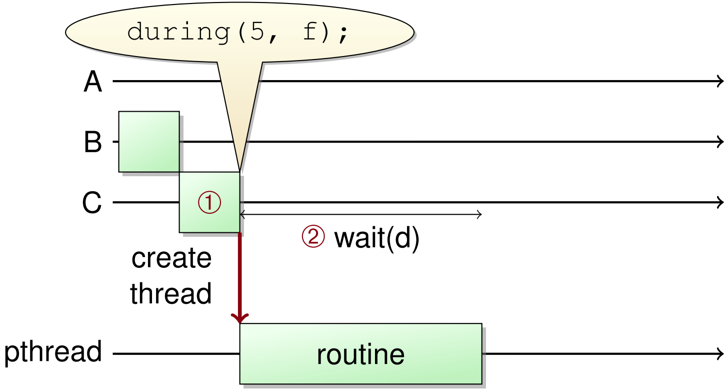
SC-DURING: Sketch of Implementation

```
void during(sc_core::sc_time duration,
         boost::function<void()> routine) {
①    boost::thread t(routine); // create thread
②    sc_core::wait(time); // let SystemC execute
③    t.join(); // wait for thread completion
}
```



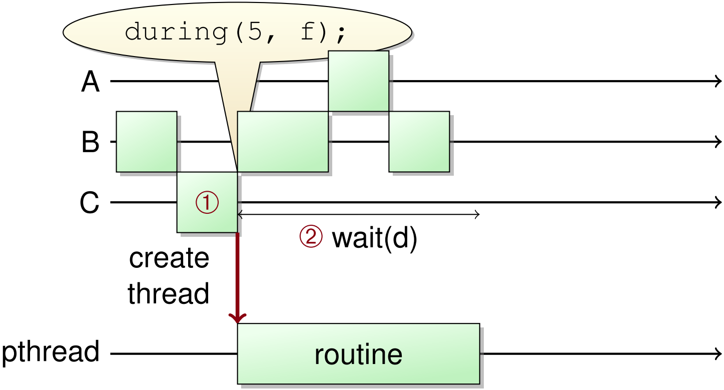
SC-DURING: Sketch of Implementation

```
void during(sc_core::sc_time duration,
         boost::function<void()> routine) {
①    boost::thread t(routine); // create thread
②    sc_core::wait(time); // let SystemC execute
③    t.join(); // wait for thread completion
}
```



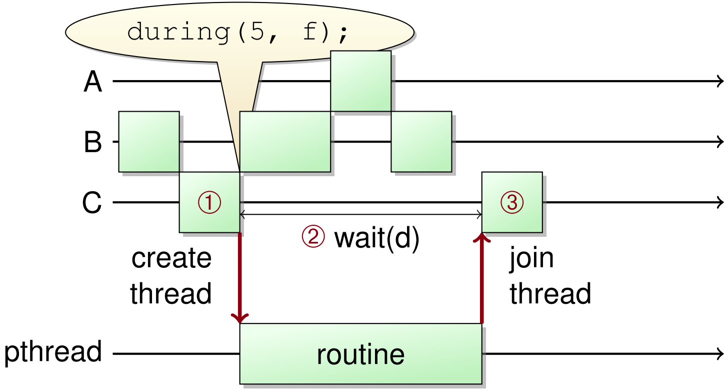
SC-DURING: Sketch of Implementation

```
    void during(sc_core::sc_time duration,
                boost::function<void()> routine) {
①       boost::thread t(routine); // create thread
②       sc_core::wait(time); // let SystemC execute
③       t.join(); // wait for thread completion
    }
```



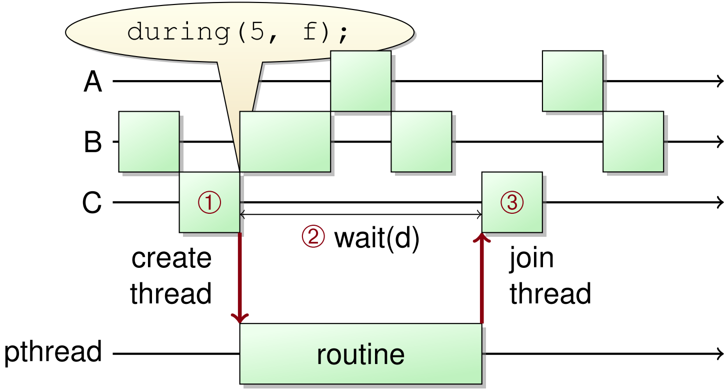
SC-DURING: Sketch of Implementation

```
    void during(sc_core::sc_time duration,
                boost::function<void()> routine) {
①     boost::thread t(routine); // create thread
②     sc_core::wait(time); // let SystemC execute
③     t.join(); // wait for thread completion
    }
```



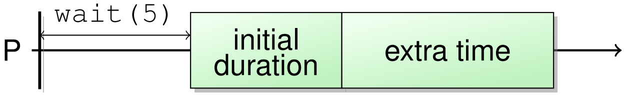
SC-DURING: Sketch of Implementation

```
    void during(sc_core::sc_time duration,
                boost::function<void()> routine) {
①      boost::thread t(routine); // create thread
②      sc_core::wait(time); // let SystemC execute
③      t.join(); // wait for thread completion
    }
```



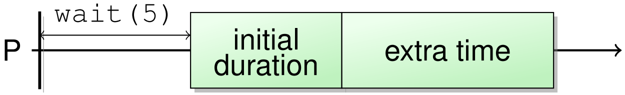
SC-DURING: Synchronization

extra_time(t): increase current task duration



SC-DURING: Synchronization

extra_time(t): increase current task duration

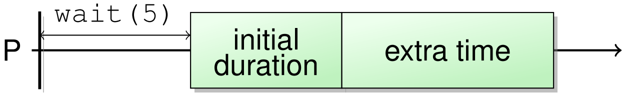


catch_up(t): block task until SystemC's time reaches the end of the current task

```
while (!c) {
    extra_time(10, SC_NS);
    catch_up(); // ensures fairness
}
```

SC-DURING: Synchronization

extra_time(t): increase current task duration



catch_up(t): block task until SystemC's time reaches the end of the current task

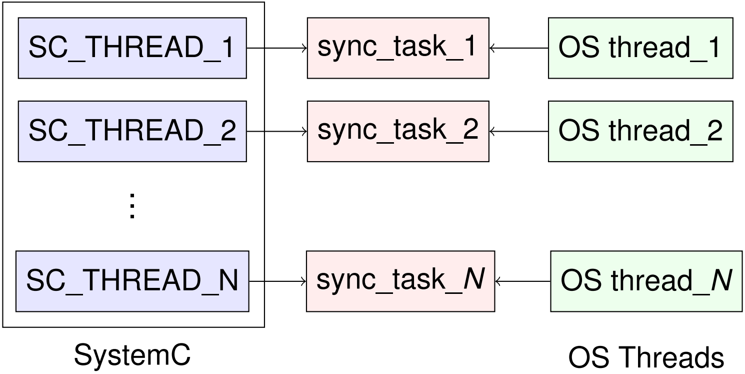
```
while (!c) {
    extra_time(10, SC_NS);
    catch_up(); // ensures fairness
}
```

sc_call(f): call function `f` in the context of SystemC

```
e.notify(); // Forbidden in during tasks

sc_call(e.notify()); // OK (modulo syntax)
```

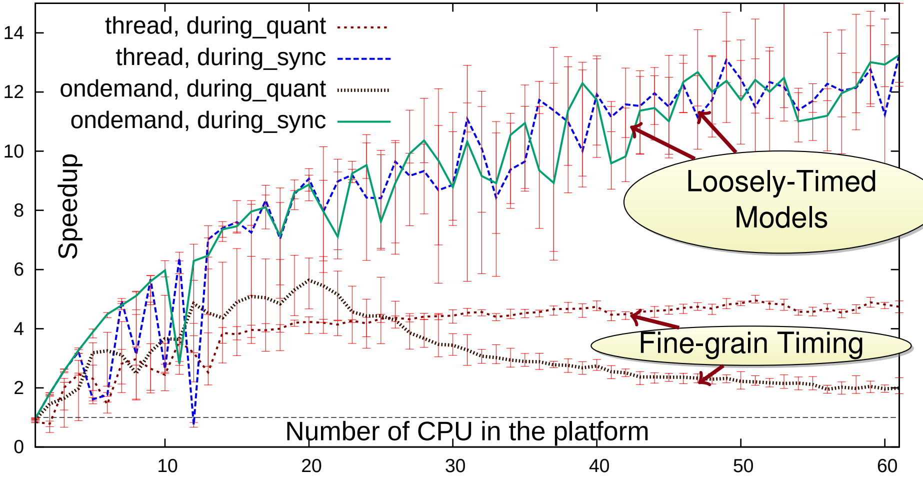
SC-DURING: Actual Implementation



Strategies:

- **SEQ** Sequential (= reference)
- **THREAD** Thread created/destroyed each time
- **POOL** Pre-allocated thread pool
- **ONDEMAND** Thread created on demand and reused later

SC-DURING: Results



Test machine has $4 \times 12 = 48$ cores

SC-DURING and jTLM: Conclusion

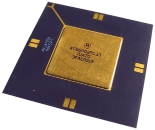
- New way to express concurrency in the platform
- Allows parallel execution of loosely-timed systems
- Exposes more bugs (⚠ faithfulness Vs correction)

This section

4 Non-functional Properties in TLM

- Time and Concurrency
  - jTLM
  - Parallelization: jTLM and SC-DURING
- Power and Temperature Estimation

Power estimation in TLM: Power-state Model

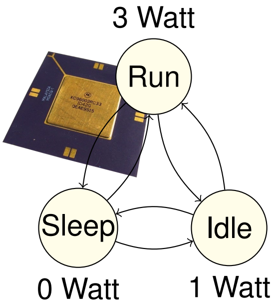



```
// SystemC thread
void compute() {
    while (true) {

        f();
        wait(10, SC_MS);

        wait(irq);
    }
}
```

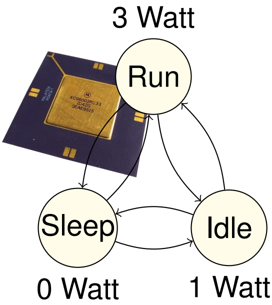
Power estimation in TLM: Power-state Model




```
// SystemC thread
void compute() {
    while (true) {
        set_state("run");
        f();
        wait(10, SC_MS);
        set_state("idle");
        wait(irq);
    }
}
```

- Consumption depends on:
  - Activity state (switching activity inside component)
  - Electrical state (voltage, frequency)

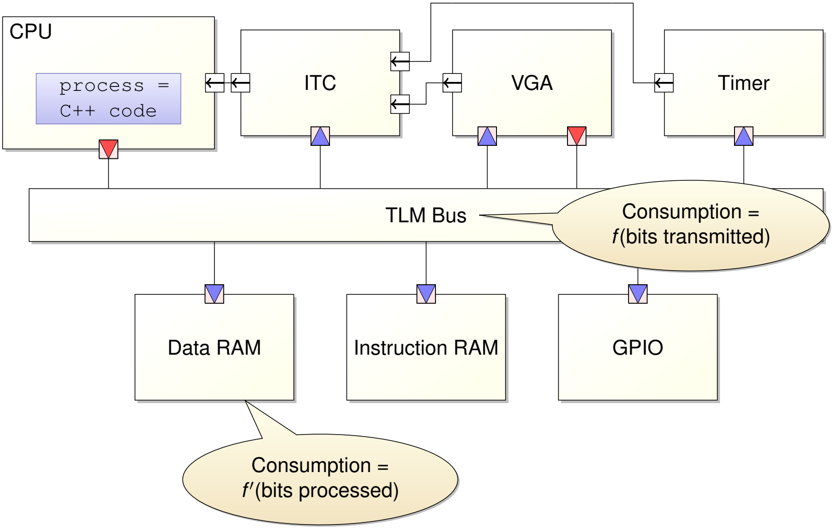
Power estimation in TLM: Power-state Model




```
// SystemC thread
void compute() {
    while (true) {
        set_state("run");
        f();
        wait(10, SC_MS);
        set_state("idle");
        wait(irq);
    }
}
```

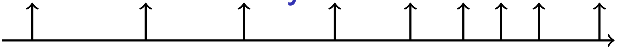
- Consumption depends on:
  - Activity state (switching activity inside component)
  - Electrical state (voltage, frequency)
  - Traffic (stimulation by other components)

Traffic Models

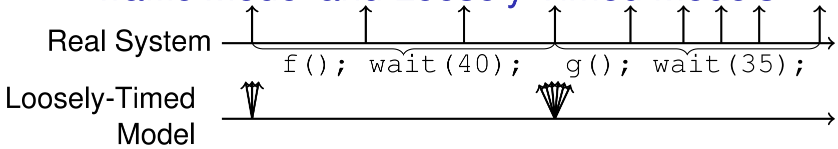


Traffic Model and Loosely Timed Models

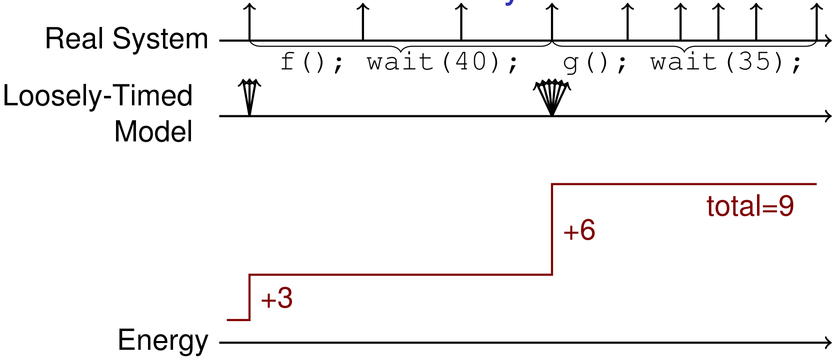
Real System



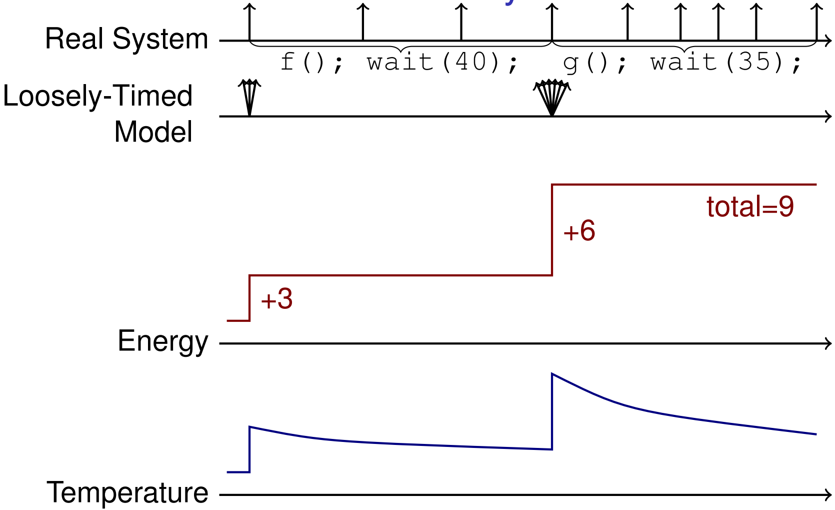
Traffic Model and Loosely Timed Models



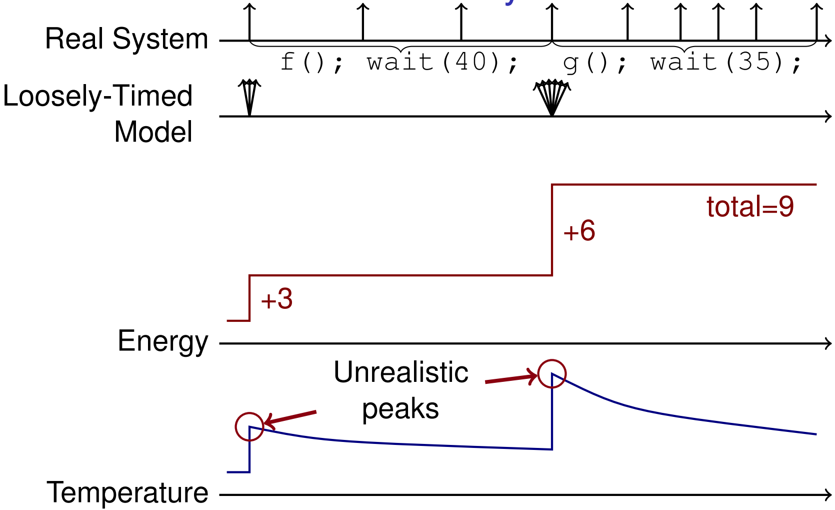
Traffic Model and Loosely Timed Models



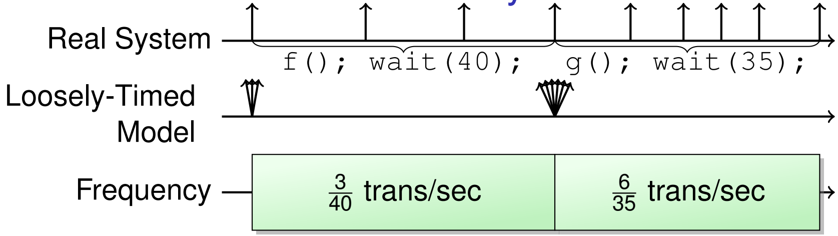
Traffic Model and Loosely Timed Models



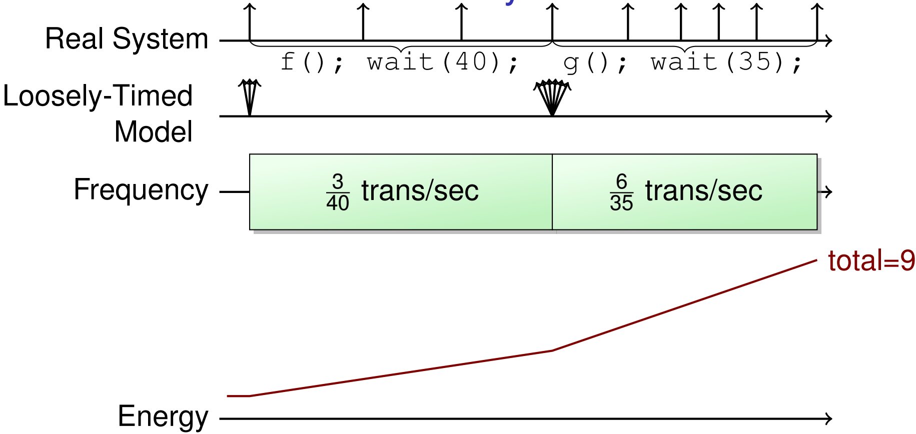
Traffic Model and Loosely Timed Models



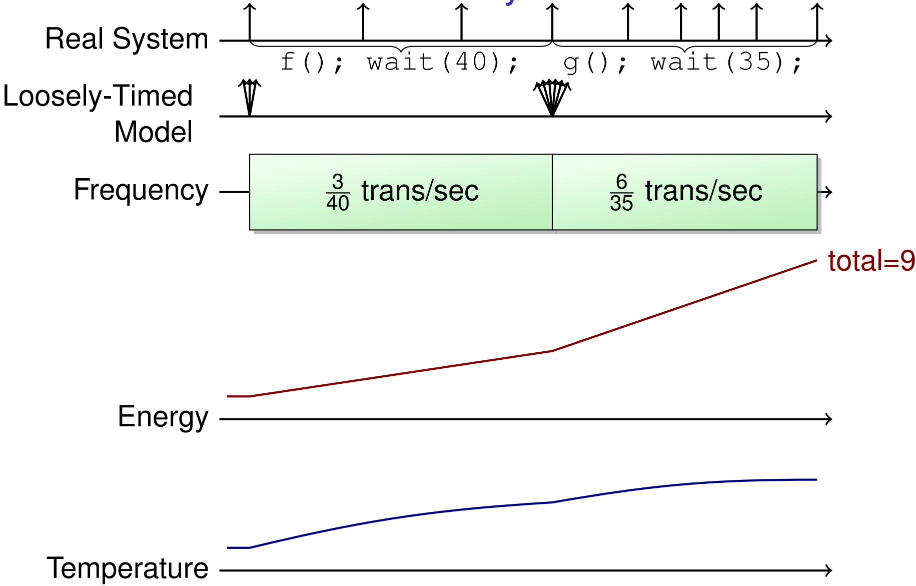
Traffic Model and Loosely Timed Models



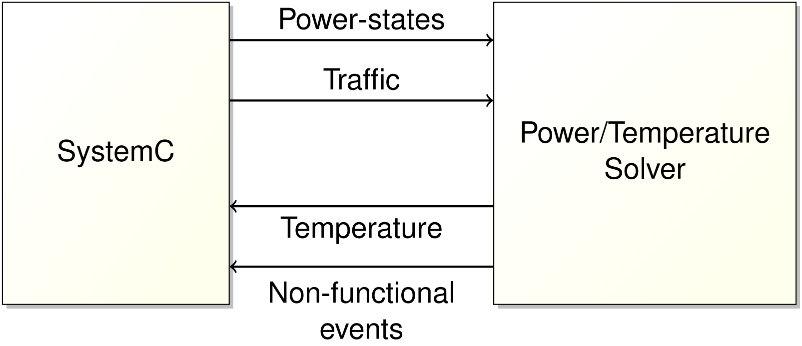
Traffic Model and Loosely Timed Models



Traffic Model and Loosely Timed Models



SystemC and Temperature Solver Cosimulation



Functionality can depend on non-functional data
(e.g. validate power-management policy)

Outline

Conclusion

Transaction-Level Models of
Systems-on-a-Chip
Can they be
Fast, Correct and Faithful?

Conclusion

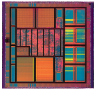
- Fast
  - Optimized compiler
  - Parallelization techniques
  - High abstraction level (Loose Timing)
- Correct
  - Formal verification
- Faithful
  - More ways to express concurrency
  - Preserve Faithfulness of Temperature Models for Loose Timing

Conclusion

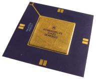
- Fast
  - Optimized compiler
  - Parallelization techniques
  - High abstraction level (Loose Timing)
- Correct
  - Formal verification
  - *Runtime Verification*
- Faithful
  - More ways to express concurrency
  - Preserve Faithfulness of Temperature Models for Loose Timing
  - *Semantics for timed systems*
  - *Refinement techniques from functional to timed models*

Questions?

Sources



`http://en.wikipedia.org/wiki/File:Diopsis.jpg`
(Peter John Bishop, CC Attribution-Share Alike 3.0 Unported)



`http://www.fotopedia.com/items/flickr-367843750`
(oskay@fotopedia, CC Attribution 2.0 Generic)