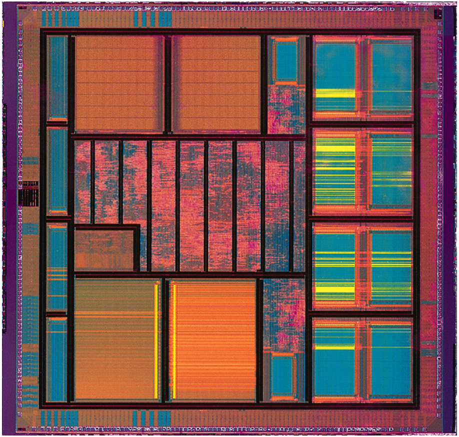
Modeling of Time in Discrete-Event Simulation for Systems-on-Chips

Matthieu Moy

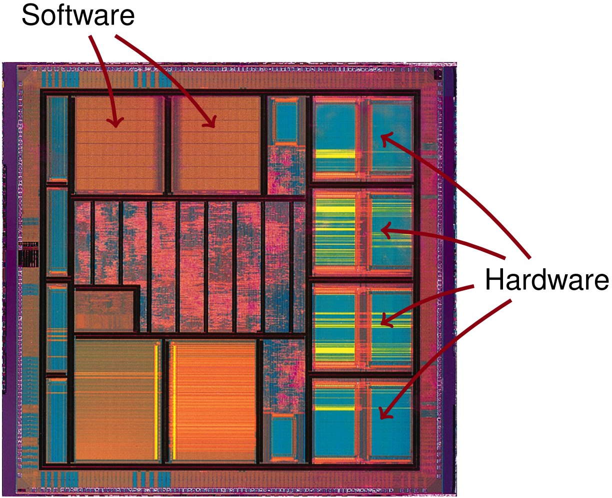
Verimag (Grenoble INP)
Grenoble, France

November 2012

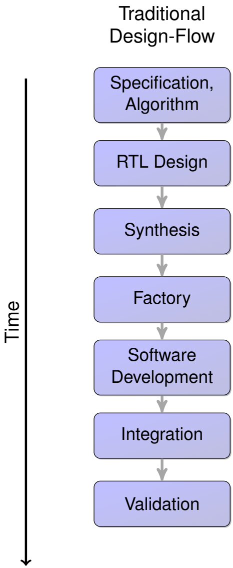
Modern Systems-on-a-Chip



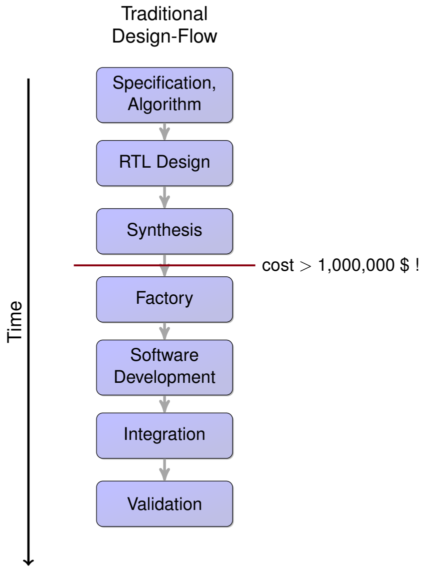
Modern Systems-on-a-Chip



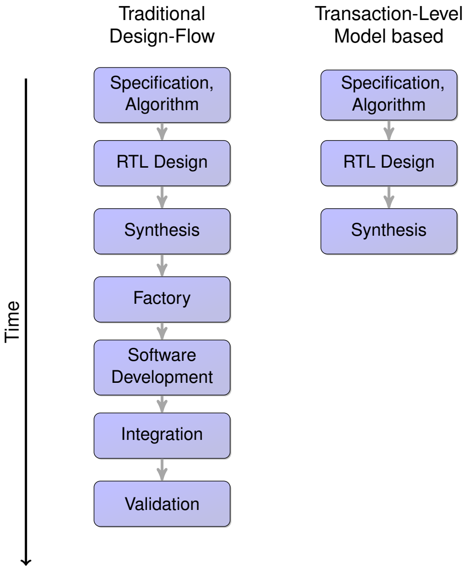
Hardware/Software Design Flow



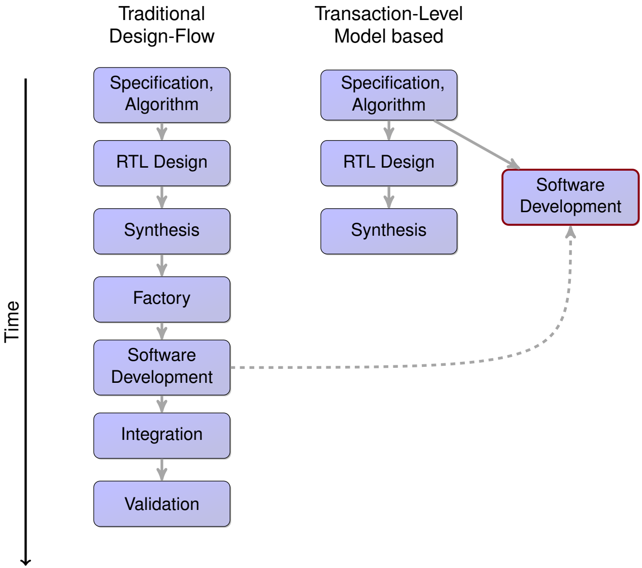
Hardware/Software Design Flow



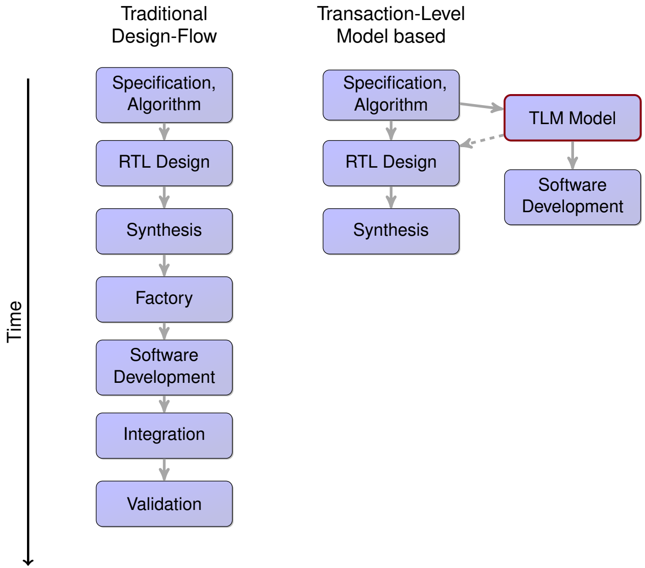
Hardware/Software Design Flow



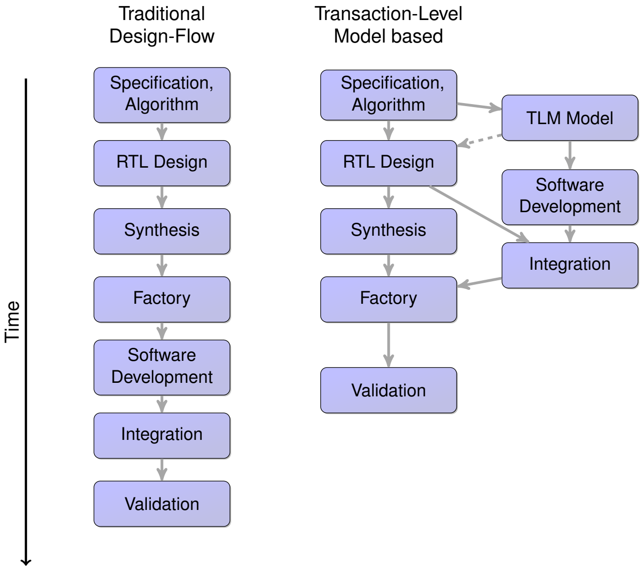
Hardware/Software Design Flow



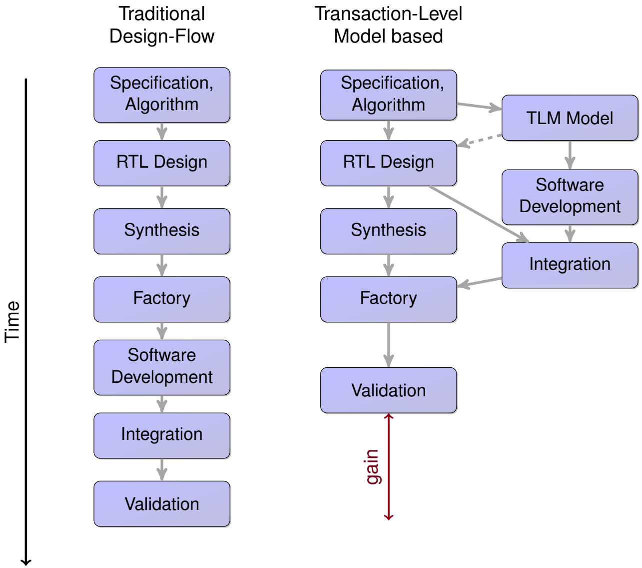
Hardware/Software Design Flow



Hardware/Software Design Flow



Hardware/Software Design Flow



Outline

Outline

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

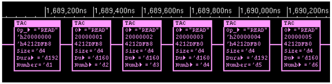
A high level of abstraction,
that appears early in the design-flow

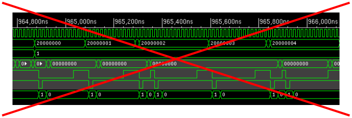
- A virtual prototype of the system, to enable
  - Early software development
  - Integration of components
  - Architecture exploration
  - Reference model for validation
- Abstract implementation details from RTL
  - Fast simulation ($\simeq$ 1000x faster than RTL)
  - Lightweight modeling effort ($\simeq$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

Content of a TLM Model

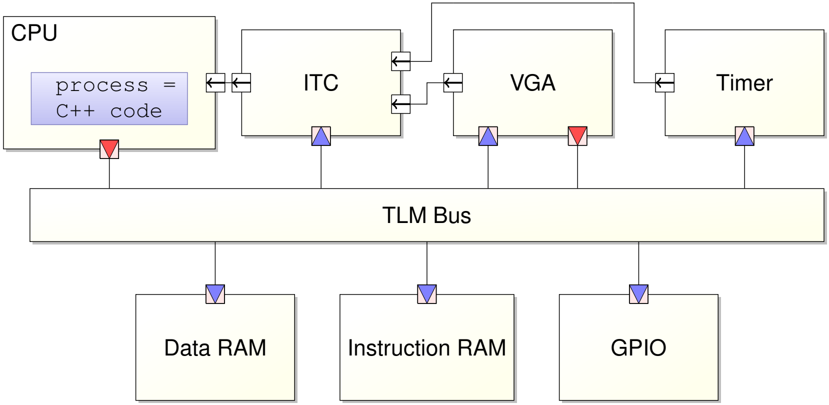
A first definition

- 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
  - ...

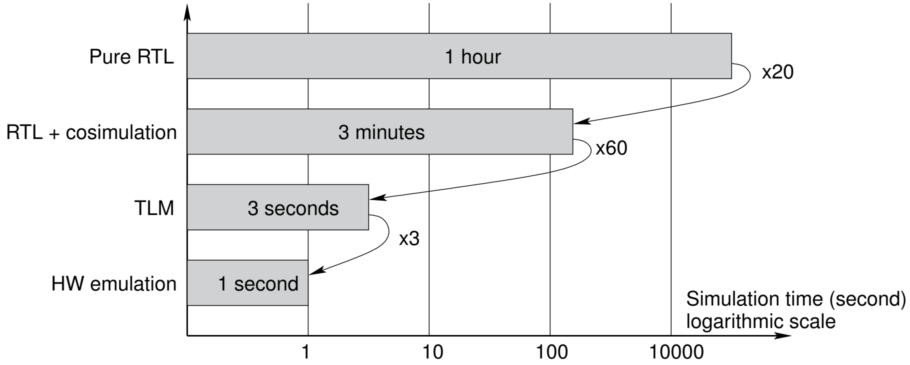




An example TLM Model



Performance of TLM



Uses of Functional Models

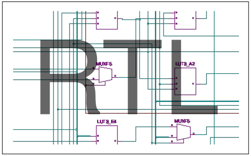
Reference for Hardware Validation

Virtual Prototype for Software Development

Uses of Functional Models

Reference for Hardware Validation

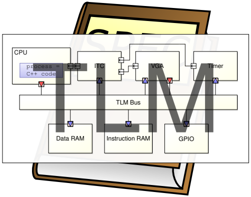
$\stackrel{?}{=}$



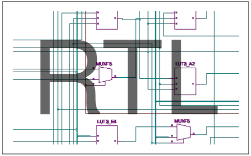
Virtual Prototype for Software Development

Uses of Functional Models

Reference for Hardware Validation



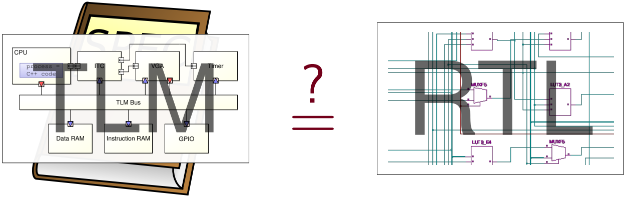
$\stackrel{?}{=}$



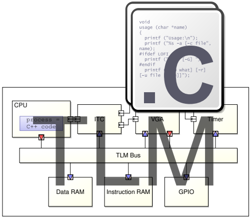
Virtual Prototype for Software Development

Uses of Functional Models

Reference for Hardware Validation

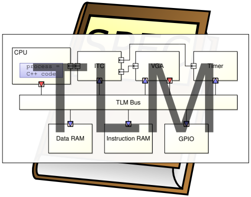


Virtual Prototype for Software Development

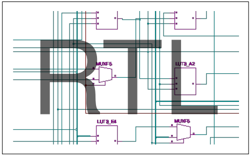


Uses of Functional Models

Reference for Hardware Validation

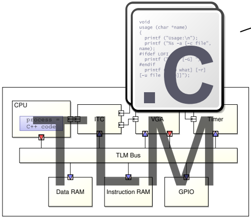


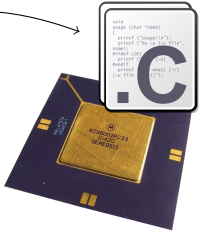
$\stackrel{?}{=}$



Unmodified Software

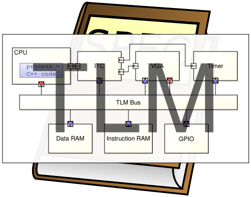
Virtual Prototype for Software Development



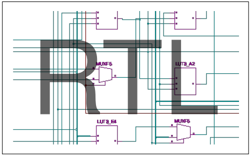


Uses of Functional Models

Reference for Hardware Validation

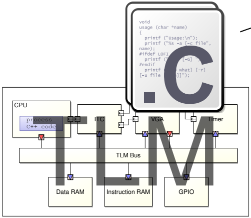


$\stackrel{?}{=}$

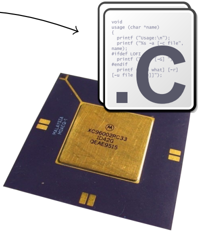


Unmodified Software

Virtual Prototype for Software Development



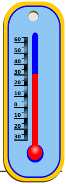
$\stackrel{?}{\supseteq}$



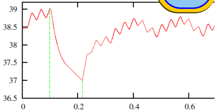
Content of a TLM Model

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

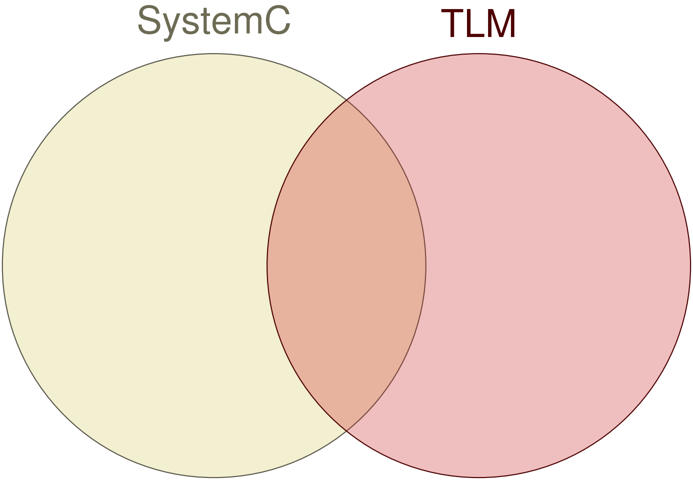


SystemC

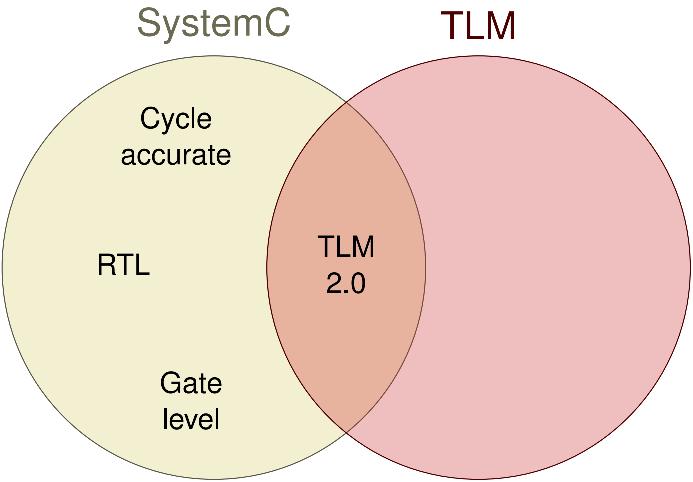
- SystemC is ...
  - a library for C++
  - a discrete-event simulator
  - well-suited for TLM
  - (an IEEE standard)
- SystemC/TLM programs are ...
  - fast (details abstracted away, efficiency of C++)
  - not fast enough (no physical parallelism)
  - too deterministic?

Outline

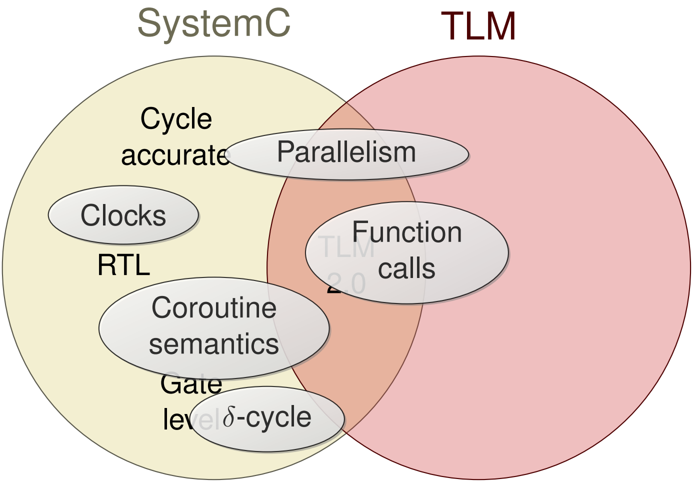
SystemC/TLM vs. "TLM Abstraction Level"



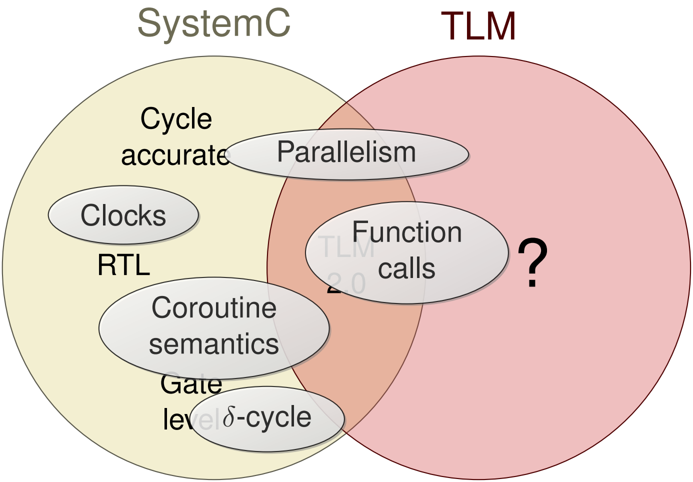
SystemC/TLM vs. "TLM Abstraction Level"



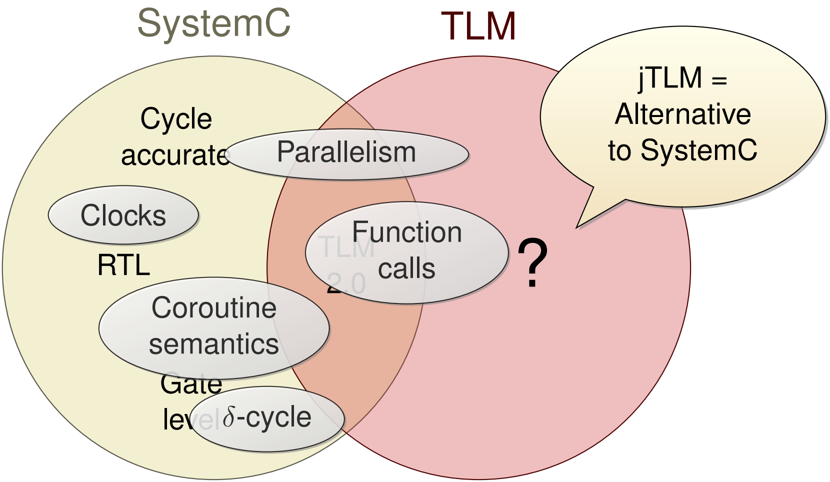
SystemC/TLM vs. "TLM Abstraction Level"



SystemC/TLM vs. "TLM Abstraction Level"



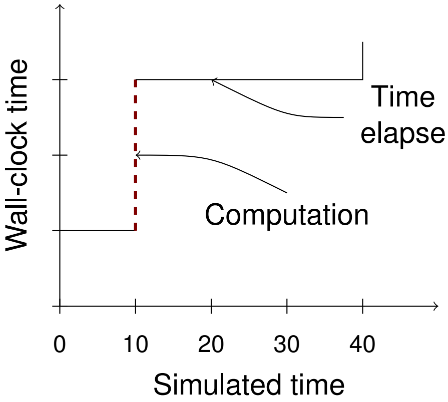
SystemC/TLM vs. "TLM Abstraction Level"



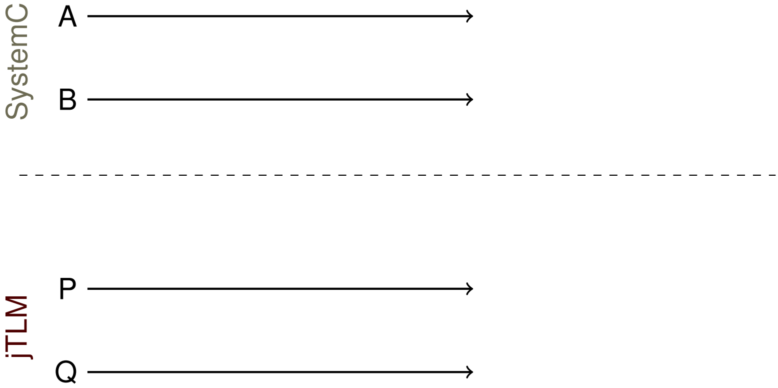
jTLM: Goals and Peculiarities

- 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

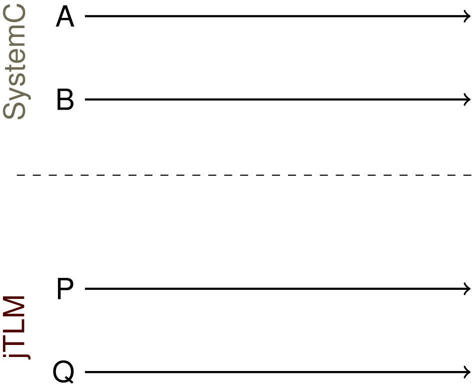
Simulated Time Vs Wall-Clock Time



(Simulated) Time in SystemC and jTLM



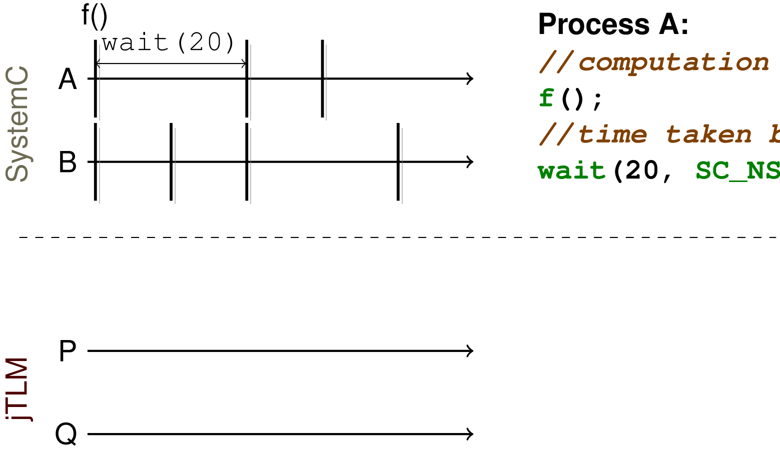
(Simulated) Time in SystemC and jTLM



Process A:

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

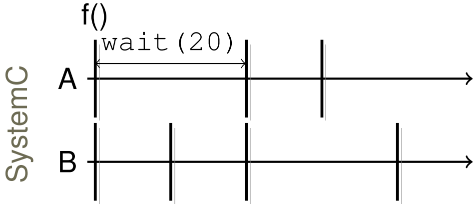
(Simulated) Time in SystemC and jTLM



Process A:

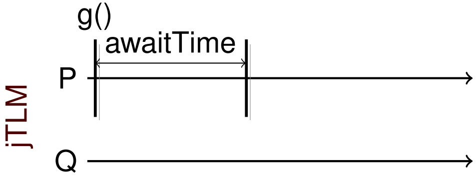
```
//computation
f();
//time taken by f
wait(20, SC_NS);
```

(Simulated) Time in SystemC and jTLM



Process A:

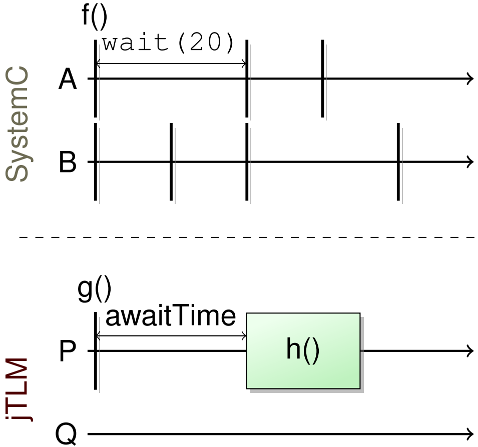
```
//computation
f();
//time taken by f
wait(20, SC_NS);
```



Process P:

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

(Simulated) Time in SystemC and jTLM



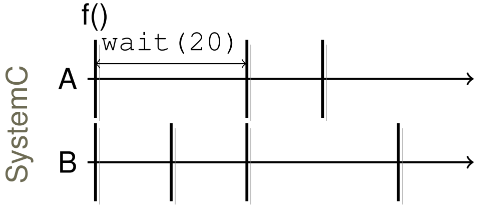
Process A:

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

Process P:

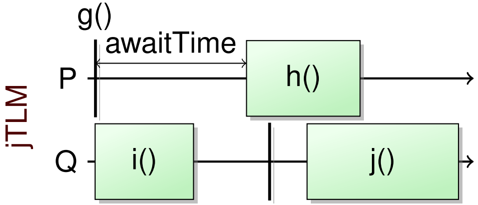
```
g();
awaitTime(20);
consumesTime(15) {
  h();
}
```

(Simulated) Time in SystemC and jTLM



Process A:

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

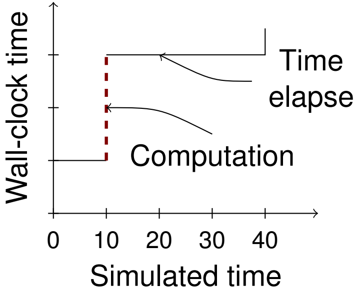


Process P:

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

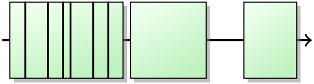
Time *à la* SystemC: `awaitTime(T)`

- 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)`

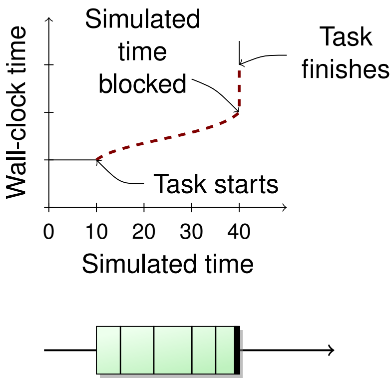


- Semantics:
  - Start and end dates known
  - Actions contained in task spread in between
- Advantages:
  - Model closer to actual system
  - Less bugs hidden
  - Better parallelization

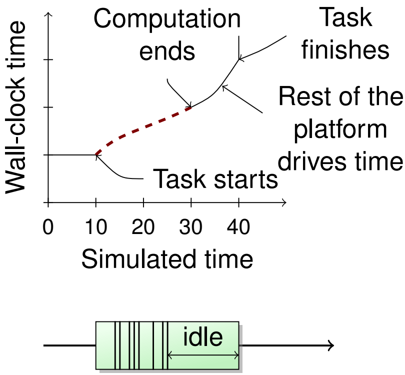
```
consumesTime(15) {
    f1();
    f2();
    f3();
}
consumesTime(10) {
    g();
}
```

Execution of `consumesTime(T)`

Slow computation



Fast computation



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:

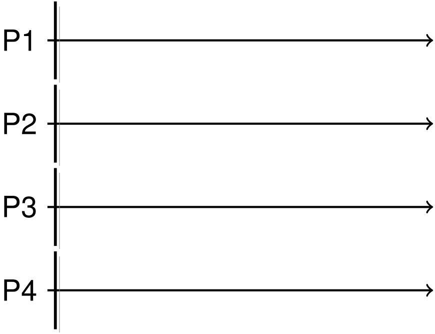
```
imgReady = true;        ||   while(!imgReady)
awaitTime(5);                    awaitTime(1);
writeIMG();                  awaitTime(10);
awaitTime(10);               readIMG();
```

⇒ bug never seen in simulation

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

⇒ strictly more behaviors, including the buggy one

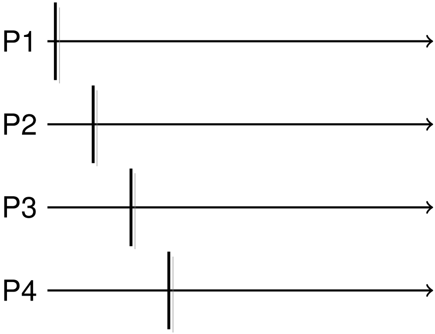
Parallelization



jTLM's Semantics

- Simultaneous tasks run in parallel

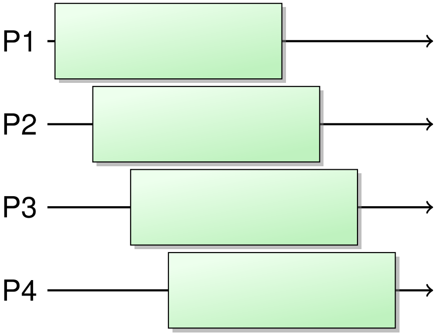
Parallelization



jTLM's Semantics

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

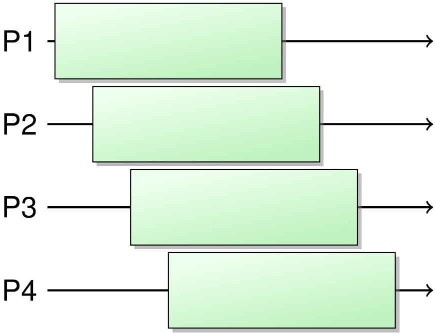
Parallelization



jTLM's Semantics

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

Parallelization

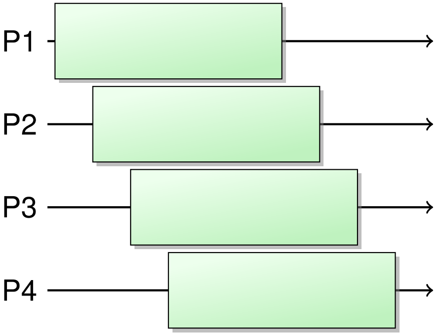


jTLM's Semantics

- 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?
(Answer in next section)

Time Queue and `awaitTime(T)`

Process P:

```
►f();
 awaitTime(50);
```

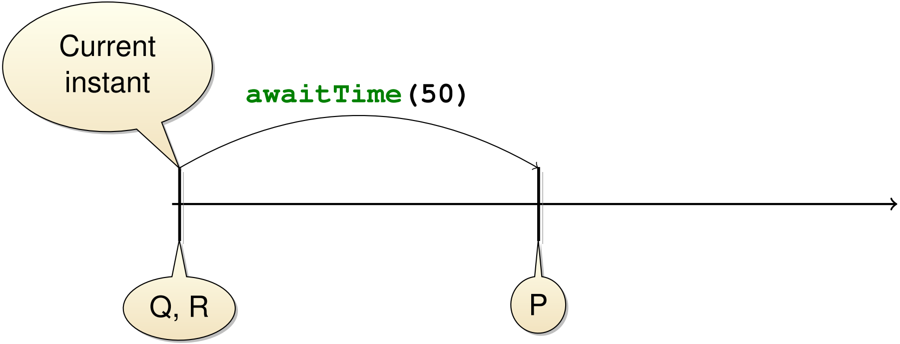
Process Q:

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

Process R:

```
►i();
 awaitTime(90);
```

Time Queue and `awaitTime(T)`

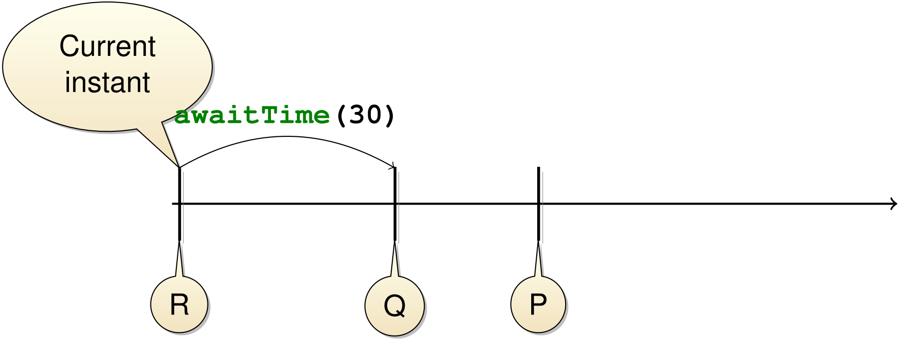


Process P:
```
 f();
▷awaitTime(50);
```

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

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

Time Queue and **awaitTime(T)**



Process P:

```
  f();
▷awaitTime(50);
```

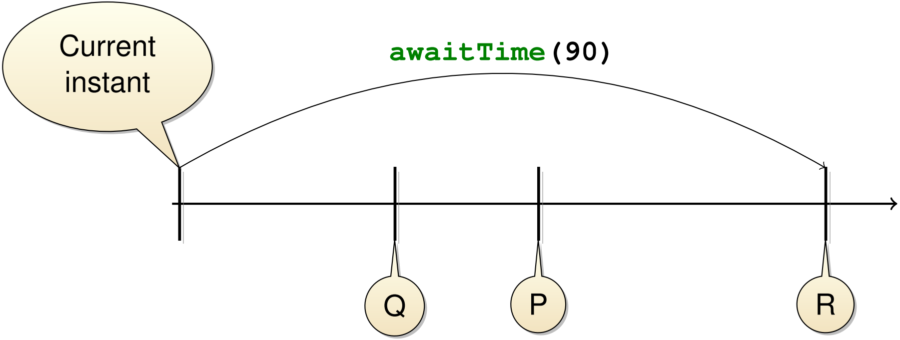
Process Q:

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

Process R:

```
▶i();
  awaitTime(90);
```

Time Queue and awaitTime(T)



Process P:

```
  f();
▷awaitTime(50);
```

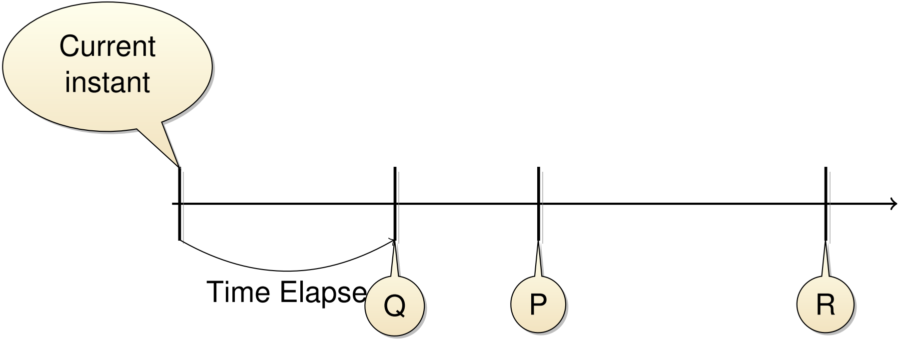
Process Q:

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

Process R:

```
  i();
▷awaitTime(90);
```

Time Queue and `awaitTime(T)`

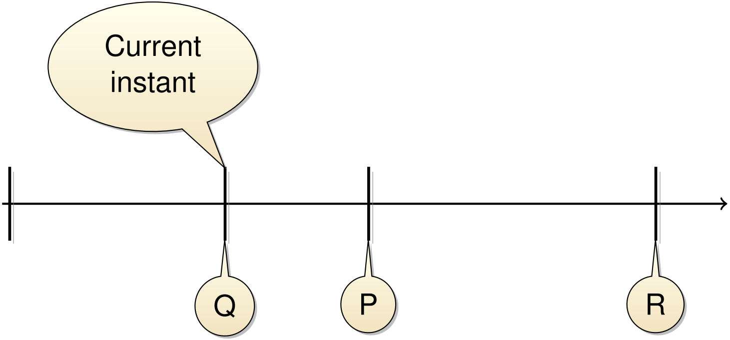


Process P:
```
  f();
▷awaitTime(50);
```

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)`?

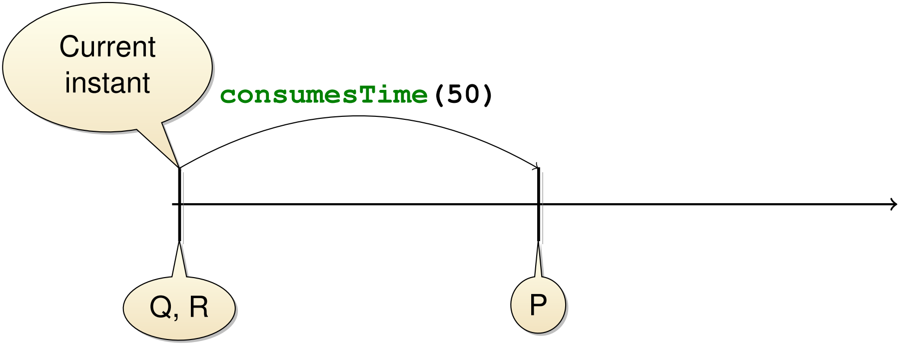
Time Queue and `consumesTime(T)`

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();
```

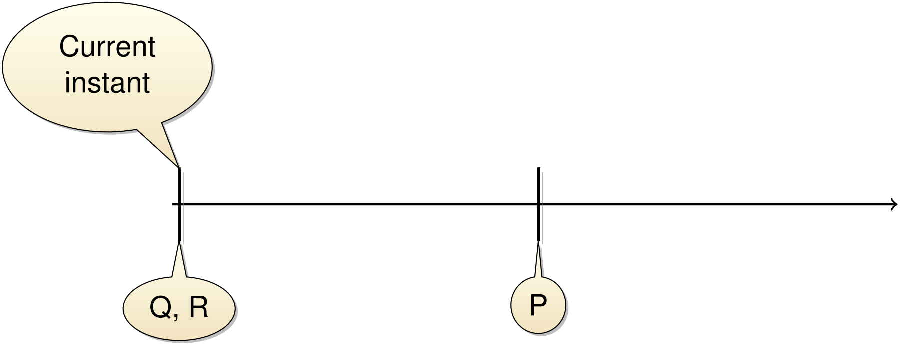
Process Q:

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

Process R:

```
►l();
 awaitTime(90);
```

Time Queue and `consumesTime(T)`

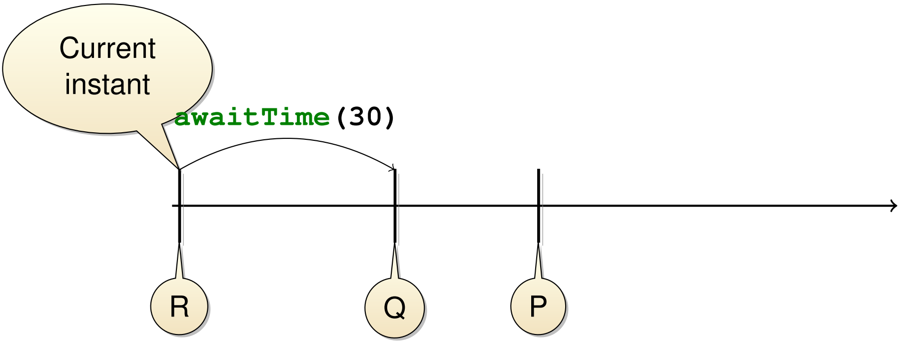


Process P:
```
 f();
 consumesTime(50){
►  g();
 }
 h();
```

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

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

Time Queue and `consumesTime(T)`

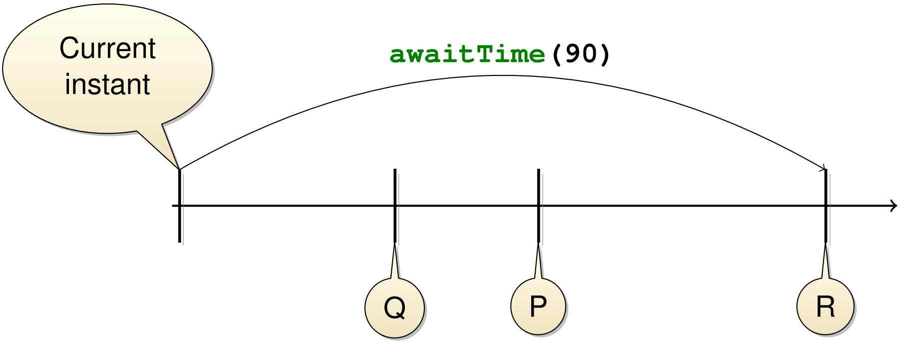



```
Process P:
  f();
  consumesTime(50){
►   g();
  }
  h();
```

```
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();
```

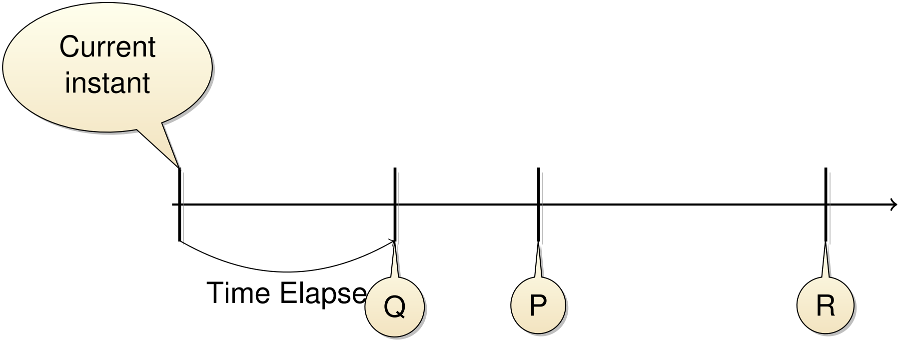
Process Q:

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

Process R:

```
 l();
▷awaitTime(90);
```

Time Queue and `consumesTime(T)`

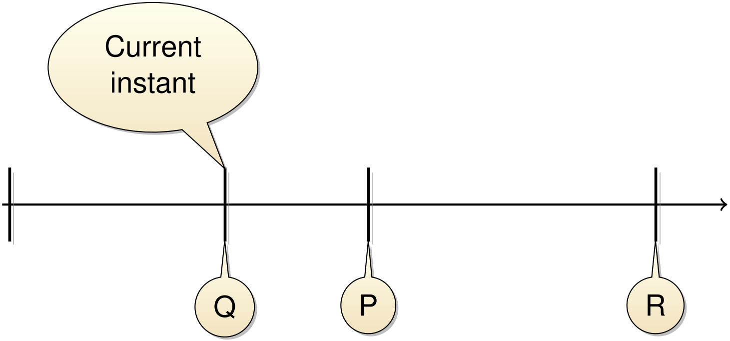



```
Process P:
  f();
  consumesTime(50){
►   g();
  }
  h();
```

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

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

Time Queue and `consumesTime(T)`

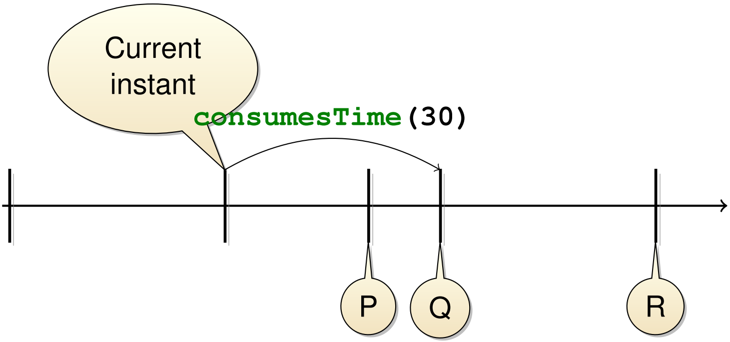


Process P:
```
  f();
  consumesTime(50){
►   g();
  }
  h();
```

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

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

Time Queue and `consumesTime(T)`

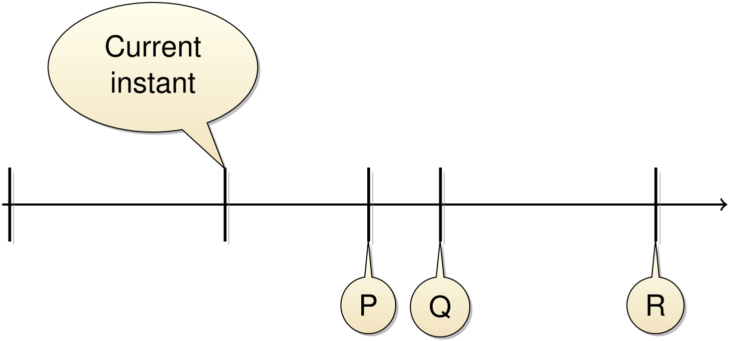



```
Process P:
  f();
  consumesTime(50){
►   g();
  }
  h();
```

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

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

Time Queue and `consumesTime(T)`

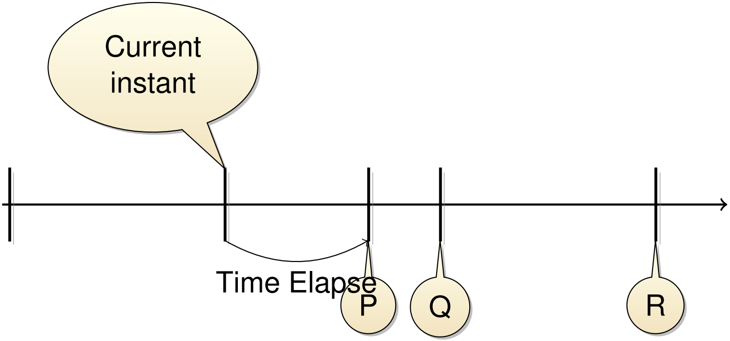


Process P:
```
 f();
 consumesTime(50){
►  g();
 }
 h();
```

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

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

Time Queue and `consumesTime(T)`

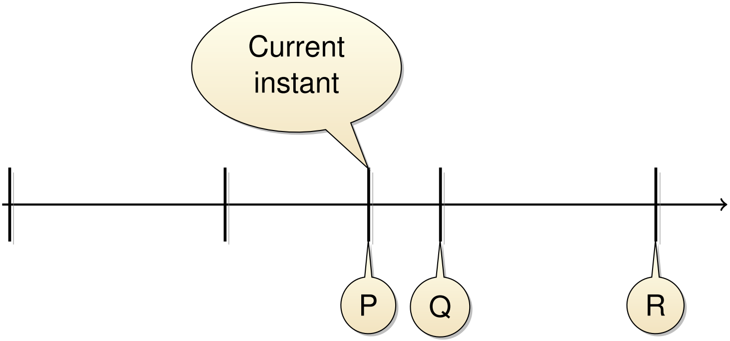


Process P:
```
  f();
  consumesTime(50){
    g();
► }
  h();
```

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);
```

Outline

jTLM is cool ...

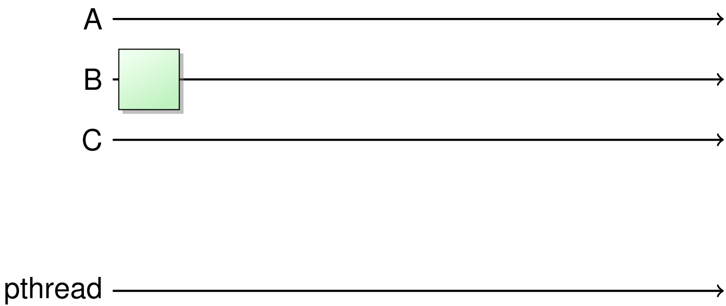
... but nobody will use it.

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

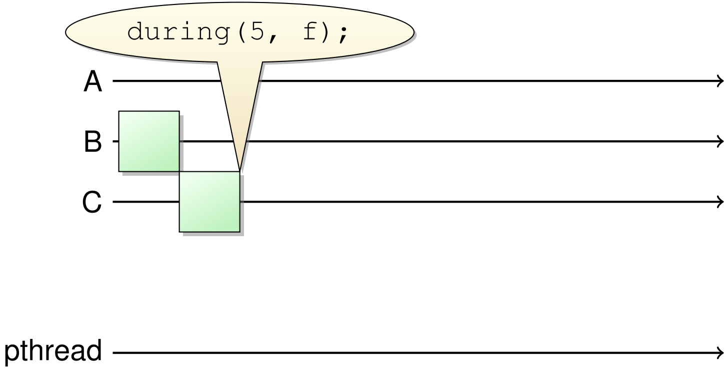
SC-DURING: Sketch of Implementation

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



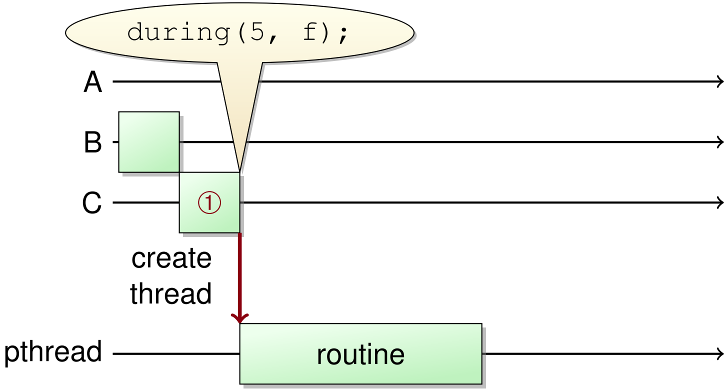
SC-DURING: Sketch of Implementation

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



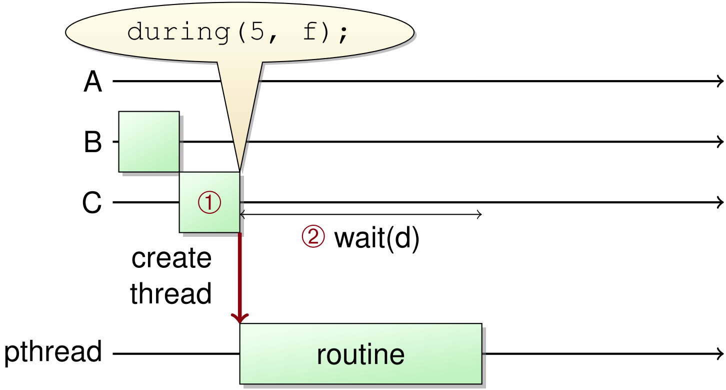
SC-DURING: Sketch of Implementation

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



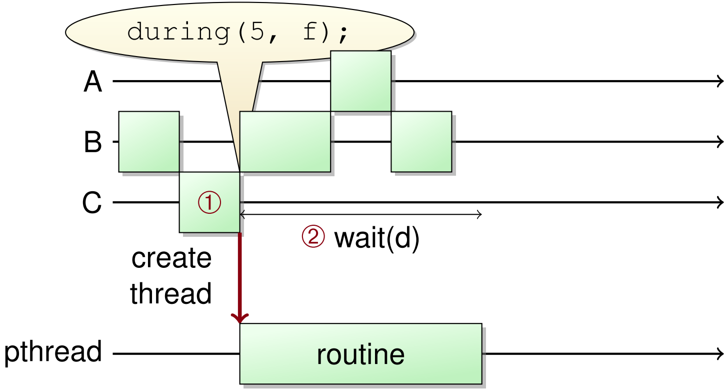
SC-DURING: Sketch of Implementation

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



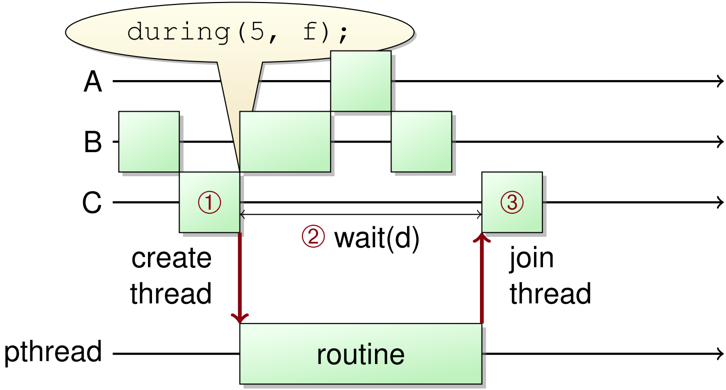
SC-DURING: Sketch of Implementation

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



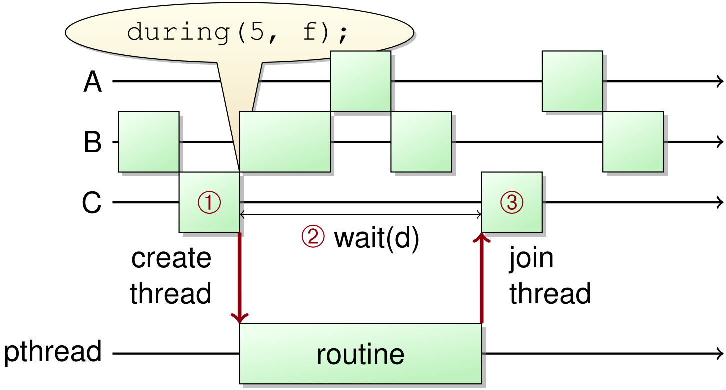
SC-DURING: Sketch of Implementation

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



Wait ... are you saying that parallelization is just about fork/join?

Wait ... are you saying that parallelization is just about fork/join?

Well, sometimes it is ...

When Things are Easy: Pure Function

Before

```
compute_in_systemc();

// my profiler says it's
// performance critical.
// does not communicate
// with other processes.
big_computation();
wait(10, SC_MS);

next_computation();
```

After

```
compute_in_systemc();

// Won't be a performance
// bottleneck anymore
during(10, SC_MS,
       big_computation);

next_computation();
```

Wait ... are you saying that parallelization is just about fork/join?

Well, sometimes it is ...

Wait ... are you saying that parallelization is just about fork/join?

Well, sometimes it is ...

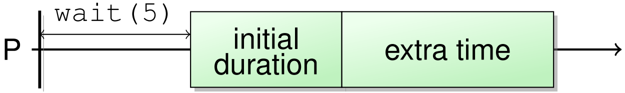
... and sometimes it isn't:

Time synchronization: make sure things are executed at the right simulated time

Data/scheduler synchronization: avoid data-race between tasks, processes and the SystemC scheduler.

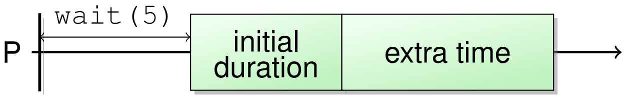
SC-DURING: Synchronization

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
}
```

EXTRA_TIME(): Sketch of Implementation

```
void during(duration, routine) {
    end = now() + duration;
    boost::thread t(routine);
    // used to be just sc_core::wait(duration)
    while (now() != end) {
        sc_core::wait(end - now());
    }
    t.join();
}
```

```
void extra_time(duration) {
    end += duration;
}
```

```
void catch_up() {
    while (now() != end) {
        // avoid busy-waiting
        condition.wait();
    }
}
```

Temporal decoupling and `sc-during`

Plain SystemC

```
f();
t_local += 42;
g();
t_local += 12;

wait(t_local);
t_local = 0;
i();
```

sc-during

```
f();
extra_time(42);
g();
extra_time(12);

catch_up();

i();
```

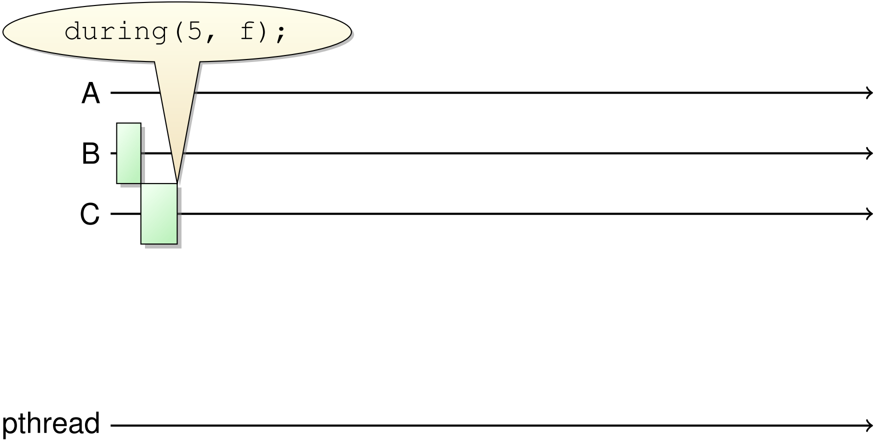
sc_call: be cooperative for a while

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

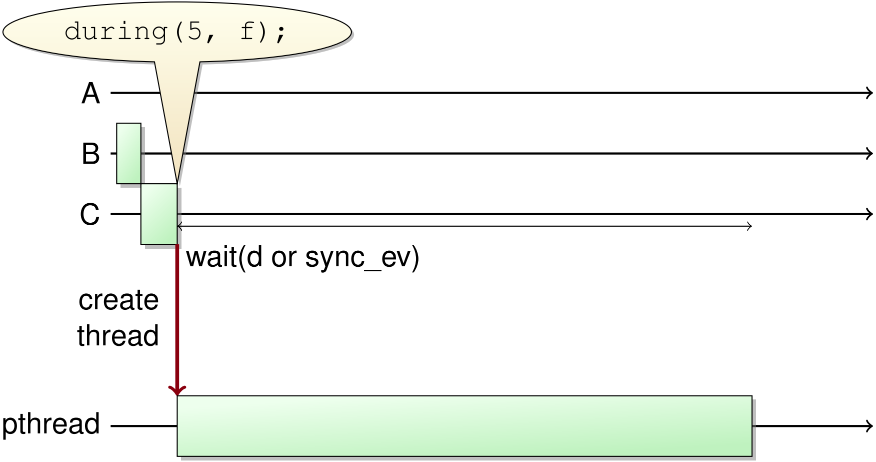
```
e.notify(); // Forbidden in during tasks

sc_call("e.notify()"); // OK (modulo syntax)
sc_call("i++"); // implicit big lock,
                // no data-race
```

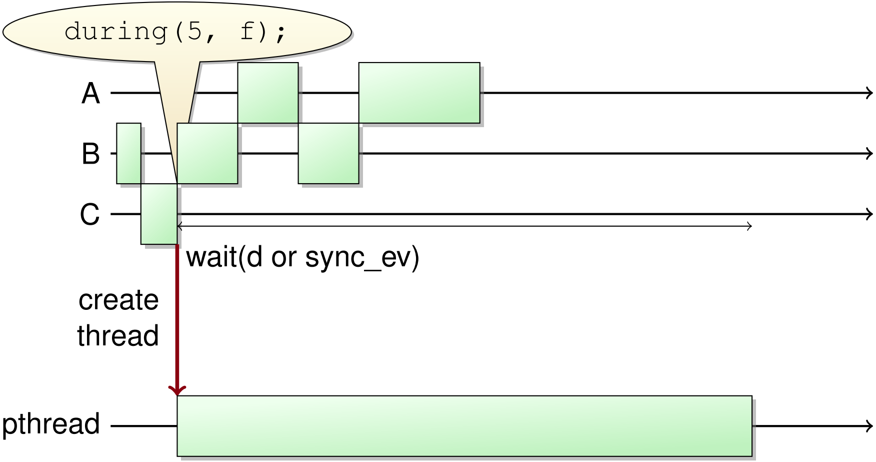
`SC_CALL`: Sketch of Implementation



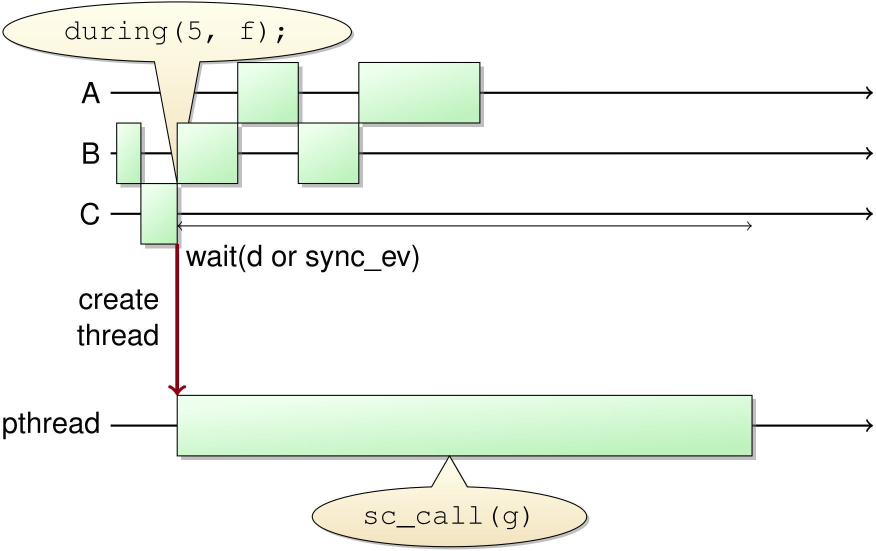
`SC_CALL`: Sketch of Implementation



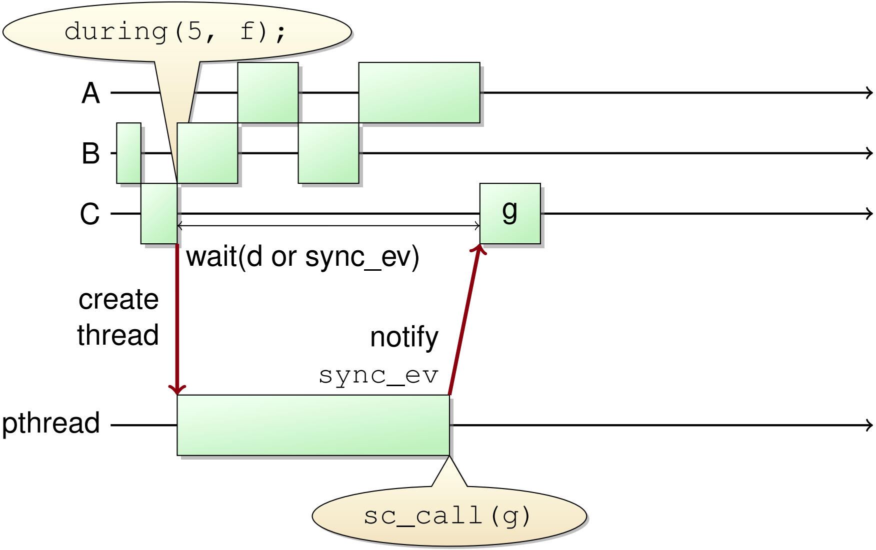
SC_CALL: Sketch of Implementation



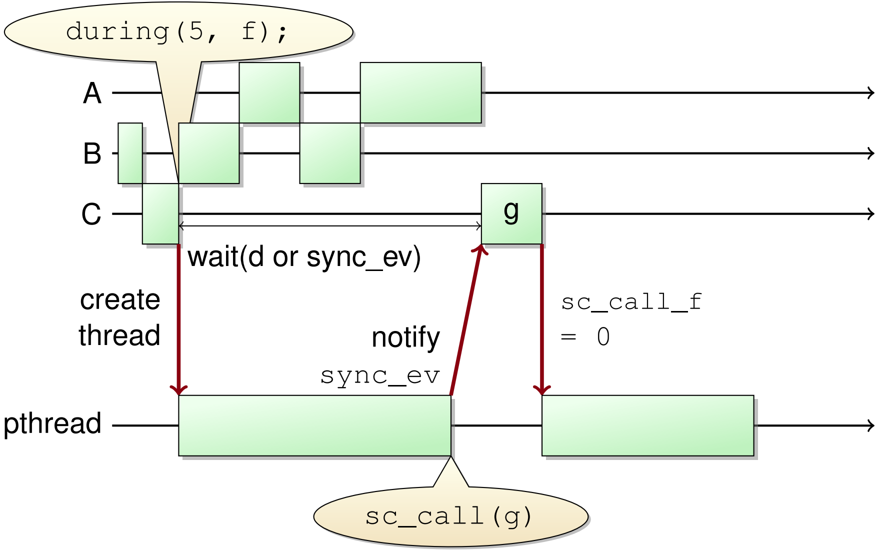
SC_CALL: Sketch of Implementation



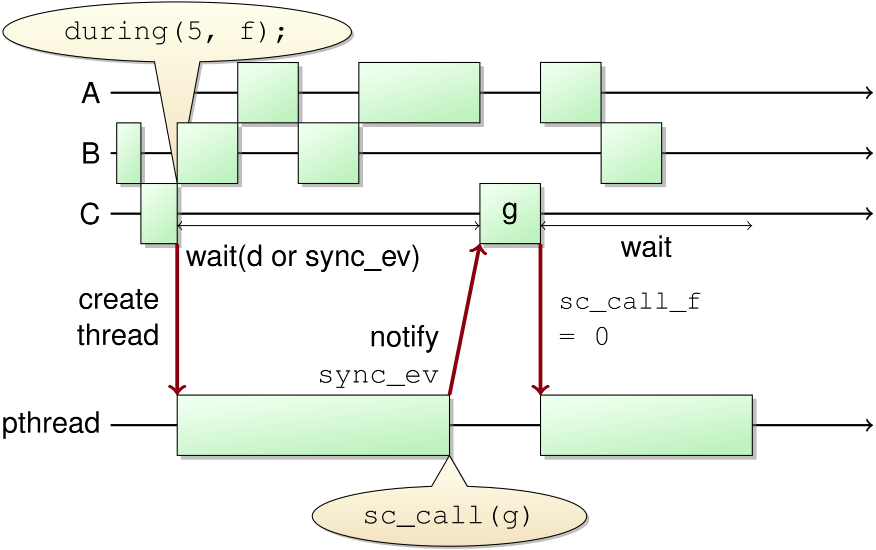
SC_CALL: Sketch of Implementation



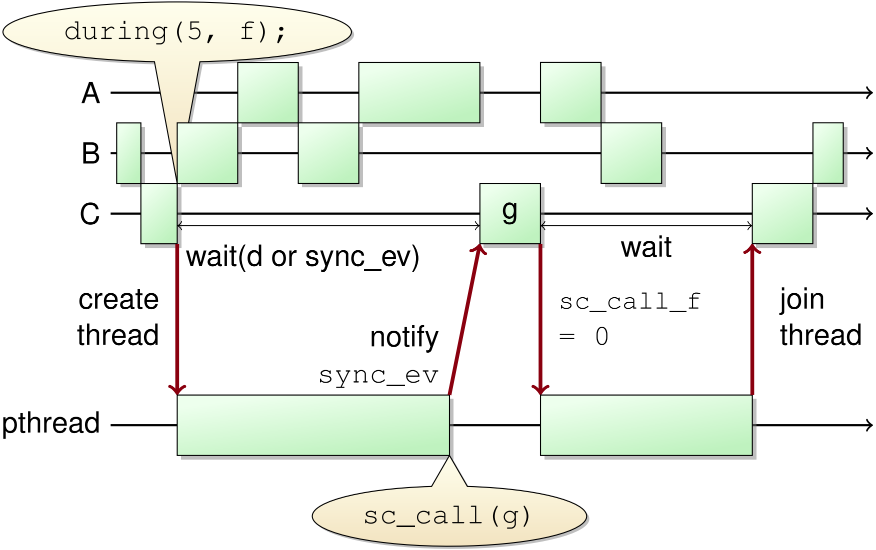
SC_CALL: Sketch of Implementation



SC_CALL: Sketch of Implementation



SC_CALL: Sketch of Implementation

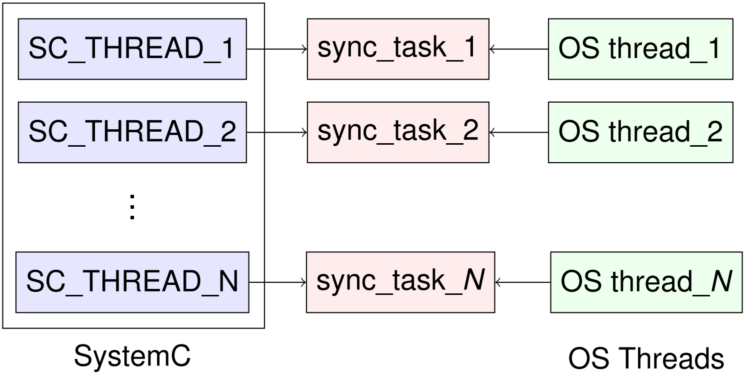


SC_CALL: Sketch of Implementation

```
void during(duration, f) {
   end = now() + duration;
   boost::thread t(f);
   while (now() != end) {
       // wait sync_ev
       // with timeout:
       sc_core::wait
          (sync_ev,
           end - now());
       if (sc_call_f) {
          sc_call_f();
          sc_call_f = 0;
          condition.notify();
       }
   }
   t.join();
}
```

```
void sc_call(f) {
   sc_call_f = f;
   // Implemented
   // with SystemC 2.3's
   // async_request_update()
   async_notify_event
       (sync_ev);
   while(sc_call_f != 0) {
      condition.wait();
   }
}
```

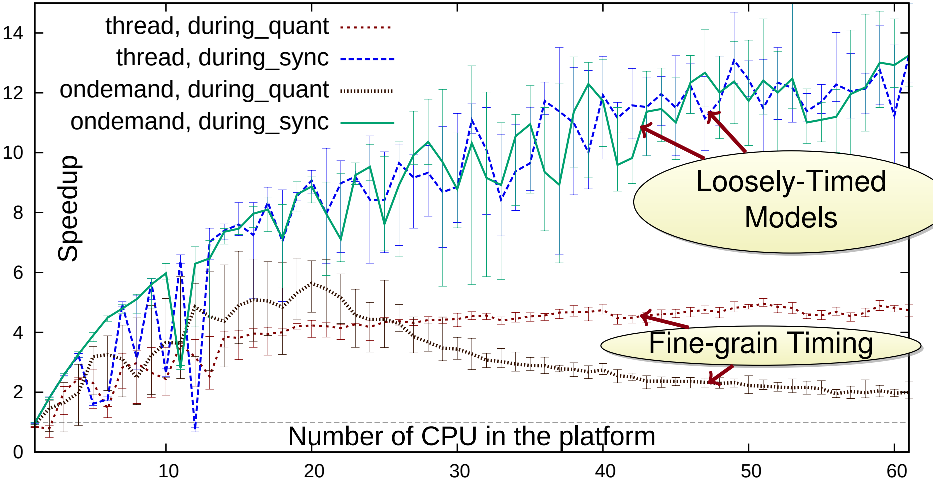
SC-DURING: Actual Implementation



Possible strategies:

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

SC-DURING: Results



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

Outline

jTLM and SC-DURING: Conclusion

- New way to express concurrency in the platform
- Allows parallel execution of loosely-timed (clockless) systems
- jTLM: experimentation platform, new scheduler
- sc-during: jTLM's ideas, implemented on top of SystemC. Still room for performance optimizations.

Try it:

`https://forge.imag.fr/projects/sc-during/`

jTLM and SC-DURING: Conclusion

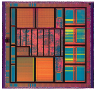
- New way to express concurrency in the platform
- Allows parallel execution of loosely-timed (clockless) systems
- jTLM: experimentation platform, new scheduler
- sc-during: jTLM's ideas, implemented on top of SystemC. Still room for performance optimizations.

Try it:

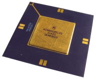
`https://forge.imag.fr/projects/sc-during/`

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)