SimTrOS: A Heterogenous Abstraction Level Simulator for Multicore Synchronization in Real-Time Systems

Jörn Schneider / Michael Bohn / Christian Eltges

Trier University of Applied Sciences
Another Simulator?
Another?

SimTrOS: A Heterogenous Abstraction Level Simulator for Multicore Synchronization in Real-Time Systems

Jörn Schneider, Michael Bohn, Christian Eltges
Dept. of Computer Science
Trier University of Applied Sciences
Trier, Germany
{j.schneider, m.bohn, c.eltges}@fh-trier.de

Abstract—To provide a common ground for the comparison of real-time multicore synchronization protocols we developed a framework that supports heterogenous levels of abstraction for simulated functionality and simulated timing. Our intention is to make the simulator available to the real-time research community. As the core of this approach, we introduce SimTrOS, a novel framework that allows researchers to select abstraction levels ranging from detailed to high-level abstractions. We demonstrate the usage of SimTrOS by evaluating the performance of several known synchronization protocols across different abstraction levels. We show that the simulator core itself can be used for any timing evaluation of multicore real-time systems and moreover, that the novel idea of heterogenous abstraction levels that lies at the heart of its design can also be a key to faster simulation through selective abstraction.
Context

• Initial purpose of the simulator
  • Evaluation of Multicore-Resource-Protocols
  • MPCP, FMLP, MSRP, ...

• At least two different aspects
  • Functional behaviour
    • Global critical sections
  • Timing behaviour
    • Which protocol performs best for a specific scenario?
  • Our scenario: AUTOSAR
• Heterogenous Abstraction Levels at work:

```java
addJob(j4) // addJob method

Ready Queue

getHighestPriorityJob() // getHighestPriorityJob method
```
• Heterogenous Abstraction Levels at work:
• Implementation 1: Sorted job list
• Heterogenous Abstraction Levels at work:
• Implementation 1: Sorted job list

\[\text{addJob}(j4) \rightarrow j6 \rightarrow j5 \rightarrow j4 \rightarrow \text{getHighestPriorityJob()} \rightarrow j1 \]

- Takes \(O(\log n)\)
- Takes \(O(1)\)
- Heterogenous Abstraction Levels at work:
- Implementation 2: Job set

```
addJob(j4) -> [j5, j6, j3, j1, j2] -> getHighestPriorityJob()
```
• Heterogenous Abstraction Levels at work:

Implementation 2: Job set

```
addJob(j4)   j4  j5  j6  j3  j1  j2
takes O(1)
```

```
getHighestPriorityJob()
takes O(n)
```
Heterogenous Abstraction Levels at work:

Which implementation to choose?
  - Implementation 1: Sorted job list
  - Implementation 2: Job set

Simulate both implementations
  - Means: implement both variants?
Teaser

- Heterogenous Abstraction Levels at work:
- Which implementation to choose?
  - Implementation 1: Sorted job list
  - Implementation 2: Job set
- Simulate both implementations
  - Means: Implement both variants?
Teaser

Functionality the same – only timing differs!
Teaser

Internal implementation

addJob(j4) → Specify different timings for operations

getHighestPriorityJob() →
getHighestPriorityJob()

Heterogenous abstraction levels

Functionality

Timing

Power consumption

...
Rest of the talk

- Usage of the simulator
- Internals of the simulator
- Conclusion
Layered architecture

Application

uses

API-calls

uses

Basic & Timing functions

uses

Simulator core
Layered architecture

Created by Users

Application

API-calls

Basic & Timing functions

Simulator core

Tasks, Interrupt-Service-Routines (ISRs), External events

Created by Users
### Defining applications

- **Task definition**

  ```
  task_i = autosarTask {
    taskPeriod = 100,
    taskPhase = 0,
    taskPriority = 1,
    taskName = "task i",
    taskCore = 0,
    taskProgram = do {
      osGetResource "R1";
      time 33;
      osReleaseResource "R1";
      time 5;
      osTerminateTask;
    }
  }
  ```

  API-calls
Defining applications

- Event definition

```plaintext
event_j = event {
    eventPeriod = Infinity,
    eventPhase = 70,
    eventName = "event_j",
    eventEffect = startISR 1 ( do {
        osActivateTask task3
    } ) interrupt on core 1
}
```
Layered architecture

Application

uses

API-calls

uses

Basic & Timing functions

uses

Simulator core

Created by Implementors

Operating system functions, helper functions
Defining API-calls

- API-call examples

```haskell
osTerminateTask = do {
    setJobVar "state" Suspended;
    schedule;
}

schedule = do {
    j <- getHighestPriorityJob;
    setRunningJob j;
}
```

Basic function calls
Layered architecture

Application

uses

API-calls

uses

Basic & Timing functions

uses

Simulator core

Primitive functions of simulation domain
Defining basic functions

- **getHighestPriorityJob**
  - **Functionality (effect)**
    - `getHighestPrioJobImpl` → pure Haskell
  - **Timing**
    - `linearTime s = 10 + 5 * length (getL readyQueue s)`
    - `constTime s = 10` → pure Haskell
Defining basic functions

getHighestPriorityJob

Functionality
getHighestPrioJobImpl → pure Haskell

Timing
linearTime \( s = 10 + 5 \)  \( \text{length } (\text{getLreadyQueue } s) \)
constTime \( s = 10 \) → pure Haskell

getHighestPriorityJob = makeBasicFunction linearTime getHighestPrioJobImpl
Simulating a system

- Compile with GHC (Glorious Glasgow Haskell Compiler)
  Application code + API-calls + Basic functions + Simulator core
- Run executable
  - Interactive (step-wise)
  - Non-Interactive
    - End of simulation
    - Time limit
- Writes XML-Logfile during simulation
Simulator core

- The core of the simulator
  - Discrete event simulation engine
  - “Hops” from event to event
  - Skips time where nothing happens
Simulator core

• Single-core example

```plaintext
task_j1 = autosarTask {
  taskPeriod = Infinity,
  taskPhase = 0,
  taskPriority = 1,
  taskName = "J1",
  taskCore = 0,
  taskProgram = do {
    time 4;
  }
}

event_isr = event {
  eventPeriod = Infinity,
  eventPhase = 3,
  eventName = "ISR",
  eventEffect = startISR 0 ( do {
    time 1;
    rf ;
  })
}
```
Simulator core

We skip this part
Simulator core

tasks

ISR

J1

0 1 2 3 4 5 6 7

time

J1 start

Interrupt
Simulator core

**tasks**

**ISR**

**J1**

**effect**: start task program J1

\( \text{effect} :: \text{SystemState} \rightarrow \text{SystemState} \)
Simulator core

tasks

 ISR

J1

time

0 1 2 3 4 5 6 7

J1 start Interrupt finish (time 4)
Simulator core

- ISR
- J1
- time
- tasks

**J1 start** | **Interrupt** | **finish (time 4)**

**effect:** activate ISR

time 4
Simulator core

- ISR
- J1
- time
- tasks

- J1 start
- Interrupt
- finish (time 1)
- finish (time 4)
- time 1
- time 4
Simulator core

tasks

ISR

J1

effect: nothing

time 1

time 4

J1 start
Interrupt
finish (time 1)
finish (time 4)
Simulator core

 ISR

 J1

 time

 tasks

 effect: return from interrupt (restore task)

 ISR

time 1 rfi

time 4

 J1 start Interrupt finish (time 4)

finish (time 1)
Simulator core

 ISR

 J1

 0 1 2 3 4 5 6 7

 tasks

 J1 start Interrupt finish (time 1)

 time

 t + (4 – 2) = 7

 (time 4) part 1

 (time 4) part 2

 (time 4) part 1

 (time 4) part 2

 time 1 rfi

 finish (time 4)
Simulator core

- ISR: time 1
- ISR: rfi
- J1: (time 4) part 1
- J1: (time 4) part 2
- J1: finish (time 4)

Tasks:
- J1 start
- Interrupt
- Finish (time 1)

Effect: nothing
Simulator core

- ISR
- J1

0 1 2 3 4 5 6 7

J1 start
Interrupt
finish (time 1)
finish (time 4)

(time 4) part 1
(time 4) part 2

time 1
rfi

(tasks)

(time 4) part 1
(time 4) part 2

finish (time 4)
Simulator core

- **tasks**
  - ISR
  - J1

- **time**
  - 0, 1, 2, 3, 4, 5, 6, 7

- **Events**
  - J1 start
  - Interrupt
  - finish (time 1)
  - finish (time 4)

- **Task Execution**
  - (time 4) part 1
  - time 1
  - rfi
  - (time 4) part 2
Simulator core

- $E =$ nearest external event
- $B_i =$ finishing time of executing basic function on core $i$

- Single-core
  - next event: $min(E, B_0)$

- Generalising to Multi-core ($n$ cores)
  - next event: $min(E, B_0, \ldots, B_{n-1})$
Simulator core

• What's with non-determinism?
  • External event effect and basic function finishes at same time
    • External event effect occurs before basic function effect
  • Two basic functions finish at same time (only multicore)
    • => User-supplied decision function called
Conclusion

• Key feature of SimTrOS
  • Separation between timing and functionality
  • Evaluate implementations that differ on timing behaviour only, without touching functional implementation

• Simulator will be available as open source:
  • Timeframe: this year

• We hope to see contributions by the community
  • Task sets
  • Operating system implementations
  • ...
Questions?

More information:

m.bohn@fh-trier.de
www.fh-trier.de/go/simtros