EFFICIENT EXECUTION AND SIMULATION OF TIME-ANNOTATED SOFTWARE

Dissertation Defense
Patricia Derler

14.10.2010
EMBEDDED SYSTEM DEVELOPMENT

Traditional Approaches:
Performance Engineering

Modern Approaches: Model based development, ...

Trade efficiency for predictability

This work: Use predictability to improving efficiency

• of execution and simulation
• without changing the behavior
PREDICTABILITY IN …

- Predictability in the Platform
  - execution time (WCET, BCET)

- Source Code
  - access to variables

- Predictability in the Environment
  e.g. temperature sensor

- Software Models
  - I/O timing

```c
T_Implementation() {
  counter++;
  ...
  if (counter % 2 == 0) {
    ...
    v2 = i2 + ...
    ...
    o1 = ...
  }
  ...
}
```
TIME-ANNOTATED SOFTWARE

T_Implementation() {
    counter++;
    ...
    if (counter % 2 == 0) {
        LET start ...
        v2 = i2 + ...
        ...
        o1 = ...; LET end
    }
    ...
}

T_Implementation() {
    counter++;
    execTime = e1; ...
    if (counter % 2 == 0) {
        execTime = e2; ...
        v2 = i2 + ...
        execTime = e2; ...
        o1 = ...;
    }
    execTime = e2; ...
}

T_Implementation() {
    counter++;
    ...
    if (counter % 2 == 0) {
        ...
        v2 = i2 ...
        ...
        o1 = ...;
    }
EFFICIENT EXECUTION ON TARGET

T_Implementation() {
    counter++;
    ...
    if (counter % 2 == 0) {
        LET start ...
        v2 = i2 + ...
        ...
        o1 = ...; LET end
    }
    ...
}
**THE LOGICAL EXECUTION TIME**

Time-annotations: logical execution times

<table>
<thead>
<tr>
<th>update inputs with outputs from sources</th>
<th>store outputs from sources such that inputs remain constant between LET start and end of the execution</th>
<th>update inputs with outputs from sources</th>
</tr>
</thead>
</table>

- read inputs
- Logical Execution Time (LET)
- write outputs

The value of each input accessed in $\delta_s$ remains unchanged between the time it is read and the start of the LET

No output value is changed after the end of the LET (in $\delta_e$)

**Runtime System**
- release tasks
- update/store outputs from sources
- update outputs at the end of the LET
BENEFITS

• Enlarging the search space for feasible schedules

• Shorter response times of lower priority event-triggered tasks

• Fewer missed deadlines of time-triggered tasks

• Increased flexibility towards adding new functionality
for every $i^{th}$ execution of a task $T$: $t_r(T)$

$$t_r(T) = \max \left\{ 0, \right.$$ 

\[ t_{Ls}(T) - \delta(T, P^S), \right. \]

\[ \left. \max_{U \in T} \left\{ t_{Le}^p(U, T) - \delta(T, P^U) \right\} \right\} \]

for every $i^{th}$ execution of a task $T$: $t_r(T)$

inputs/outputs implemented as global variables

further input:
- minimum preemption time
- inputs from event-triggered tasks
TASK TERMINATION TIME

\[
t_t^i(T) = \min \left\{ t_{i+1}^i(T), \min_{q \in P_{OUT}(T)} \left\{ t_{Le}^i(T) + \delta(q, T) \right\} \right\}
\]
SCHEDULING SUSTAINABILITY

A system that is schedulable by EDF with the classical LET constraints is also schedulable by EDF with the flexible scheduling constraints.

follows from sustainability of EDF shown in


A system that is schedulable by FP with the classical LET constraints is not necessarily schedulable by FP with the flexible scheduling constraints.

If a job set is schedulable by a preemptive policy, then the set of modified jobs is schedulable by any preemptive policy which introduces no new preemption points in the execution intervals.
SUSTAINABLE SCHEDULING

Solution 1:
Early release times only up to the end of the LET of any preceding higher priority task

Solution 2:
Dual Priority Scheduling

Dual priority = lower than all nominal priorities of all other jobs
EVALUATION

- Find examples of systems that are unschedulable with classical scheduling constraints and can be made schedulable with flexible constraints

- Flexibility of a system towards adding new functionality
  - to a task
  - in a new task

classical:

flexible:
**EVALUATION**

- Response time of event-triggered tasks

Simulation of FP and EDF schedule with classical and flexible scheduling constraints

Mean decrease of response time of ~25%
EFFICIENT SIMULATION ON HOST

T_Implementation() {
    counter++;  
    execTime = e1; ...
    if (counter % 2 == 0) {
        execTime = e2; ...
        v2 = i2 + ...
        execTime = e2; ...
        o1 = ...
    }
    execTime = e2; ...
}
ACCESS POINTS

An access point (AP) is a line of source code with a reference to a global variable.

```
T_Implementation() {
  ...
  v = ...
  ...
  ... v' ...
  ...
  ...
}
```

An access point event (APE) is generated before executing an AP.

The execution of software is controlled at APs.
EXAMPLE: EXECUTE AND SIMULATE 2 TASK EXECUTIONS

Execution on target

Simulation with 0 execution time

Simulation with execution time from target

T\_Implementation() \{ \text{...} = \nu \text{...} \}

T’\_Implementation() \{ \text{...} \nu = \text{...} \text{...} \}

\[ \delta_1 \]

\[ \delta_2 \]

\[ \delta_3 \]

\[ \delta_4 \]

\( T \)

\( T' \)

\( t_0 \)

\( t_1 \)

\( t_2 \)

\( t_3 \)

\( t_4 \)

\( \delta_1 - (t_4 - t_3) \)

preempted

\( \delta_2 \)

\( \delta_3 \)

\( \delta_4 \)

\( \delta_1 - (t_1 - t_0) \)

different behavior!

\( T' \)

\( T \)

\( t_0 \)

\( t_1 \)

\( t_2 \)

\( t_3 \)

\( t_4 \)

\( \delta_1 - (t_4 - t_3) \)

\( \delta_2 \)

\( \delta_3 \)

\( \delta_4 \)

\( \delta_1 - (t_1 - t_0) \)

same behavior
The execution of software has to be delayed at access points $\alpha^T(v, \delta)$ if during the next $\delta$ time units a preemption can occur where another task accesses $v$. Tasks have to access variables in the same order, not necessarily at the same times.
CLOSED-LOOP SIMULATION

Software

\[ v = \ldots \]

\[ \text{T\_Implementation()} \{ \]
\[ \ldots = v \ldots \]
\[ v = \ldots \]
\[ \} \]

Plant

Software simulation time

Plant simulation time

Synchronization whenever data is exchanged

Before executing an access point \( \alpha^T(v, \delta) \), where \( v \) is also accessed by the plant, model times have to be synchronized
EQUIVALENT EXECUTIONS

Definition: Two executions E and E’ of the same software are equivalent if

• task executions in E are also contained in E’ and vice versa
• every task execution reads the same input values and writes the same output values

APE trace: \[ AT = \{ (\alpha^T_i (v, \delta), t_i) \}_{i \in \mathbb{N}}, T \in T, v \in V \]

Projection of a variable v: \[ AT(v) = \{ (\alpha^T_j (v, \delta), t_j) \}_{j \in \mathbb{N}}, T \in T \]

Theorem: Two executions E and E’ are equivalent if the variable projections
\[ AT(v) = \{ (\alpha^T_j (v, \delta), t_j) \}_{j \in \mathbb{N}}, T \in T \]
and \[ AT(v)' = \{ (\alpha'^T_j (v, \delta), t'_j) \}_{j \in \mathbb{N}}, T \in T \]
satisfy the following conditions

• tasks access variables in the same order \[ T = T' \quad \forall \alpha^T_j \in AT(v), \forall \alpha'^T_j \in AT(v)’ \]
• variables that can be accessed by the plant are accessed at the same time in both executions \[ t_j = t'_j \quad \forall v \in V_H \]
EXECUTION CONTROL

The execution controller decides at every access point whether the execution can continue or has to be delayed

- C1: always continue execution = pure functional simulation

- C2: always delay execution

- C3: delay execution at access points if there exists a task with a higher priority that can access the same variable

- C4: lookahead execution control delay at APs if a task can be triggered before the AP is executed which modifies the same resource.

Preserve execution semantics

simulation cost:
- fewer task switches
- fewer synchronization points between software and plant
IMPLEMENTATION

T_Implementation() {
    counter++;
    ...
    if (counter % 2 == 0) {
        ...
        v2 = i2 + ...
        ...
        o1 = ...
    }
    ...
}

T_Implementation() {
    counter++;
    ...
    if (counter % 2 == 0) {
        ...
        execTime = e1;
        if (counter % 2 == 0) {
            ...
            execTime = e2;
            v2 = i2 + ...
            ...
            execTime = e2;
            o1 = ...
        }
        ...
    }
    ...
}

Execution time analysis on basic block level

Legacy Code

+ Execution time annotations

+ Callbacks to Execution Controller before access points

---

Vehicle Model

exhaust
vehiclespeed
crank90
N
manifold pressure
 crank180

brake
throttle
fuel
gear
spark

Plant Model

---

Platform Model

Interrupt Service Controller

Timer

Operating System Model

CPUScheduler
EventManager
Resources

Software Model

---

Execution Controller

21
EVALUATION

Vehicle Model
- brake
- throttle
- fuel
- gear
- spark
- exhaust
- vehiclespeed
- crank90
- manifold pressure
- crank180

User Input
- brake inc
dec
resume
trg
acceleration
power
set
gearOn

Throttle Controller
- NE [rad/sec]
throttle command
cruiseOn

EnableISC

Spark Angle Controller
- manifold pressure
spark angle

Idle Speed Controller
- desired speed
throttle angle

Gear shift Controller
- throttle
engage
gear

Fuel Rate Controller
- manifold pressure
throttle
fuel
gear
spark

Cruise Controller
- current speed
brake increment
decrement
resume trigger
pedal acceleration
power
set
cruise on
target speed

Time-triggered Task

Event-triggered Task

Plant

Software

User Input

Vehicle Model

Time-triggered Task

Event-triggered Task

Software
EVALUATION

Academic Example: Simulation of 100 seconds

C1: always continue execution
C2: always delay execution
C3: delay execution at access points if there exists a task with a higher priority that can access the same variable
C4: lookahead execution control

C2 ~4x more delay events than C1

C1 ~4x faster than C2

Industrial Application: Simulation of 500ms

C2 ~35x more delay events than C1
C4 ~5x more delay events than C2

C1 ~55x faster than C2
C4 ~9x faster than C2
ADD LOGICAL EXECUTION TIMES TO LEGACY CODE

T_Implementation() {
    counter++; 
    ...
    if (counter % 2 == 0) {
        LET start; ...
        v2 = i2 + ...
        ...
        o1 = ... LET end
    }
    ...
}

LET Execution Controller

Execution Controller

Compare software with and without LETs
SUMMARY

**Efficient execution** on a target platform
- use predictability to compute
  - release times $< \text{LET starts}$
  - termination times $> \text{LET ends}$
- for LET-based tasks
- to maximize scheduling flexibility

**Efficient simulation** on a host platform
- by controlling software execution and plant simulation at well-defined steps (access points)
- and using predictability to reduce amount of
  - task switches and
  - synchronization steps between plant simulation and software execution
- to minimize simulation duration

while preserving the original behavior