Semantics-Based Compiler Transformations for Enhanced Schedulability

Richard Gerber & Seongsoo Hong

Presented by Li Deng

Main idea

- Using TCEL, a real-time programming language, the unobservable code can be automatically moved, so, an unschedulable task set can be convert into a schedulable one

Outline

- Introduction
- Overview of TCEL
- Scheduling with Compiler Transformations
- Automatic Task Decomposition by program Slicing
- Conclusion

Introduction—the TCEL language

- TCEL — Time-Constrained Event Language
- Compare with other languages:
  - Other languages establish constraints between blocks of code
  - TCEL semantics establishes constraints between the observable events within the code

Introduction—the TCEL language

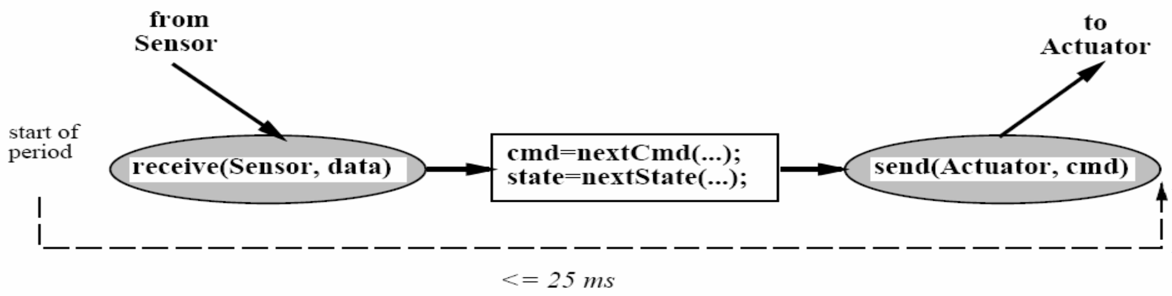


Figure 1: Structure of Controller Subsystem.

TCEL program fragment:

```
A1:  every 25ms
        {
A2:     receive(Sensor, data);
A3:     cmd = nextCmd(state, data);
A4:     state = nextState(state, data);
A5:     send(Actuator, cmd);
        }
```

```
A1:  every 25ms
        {
A2:     receive(Sensor, data);
A3:     cmd = nextCmd(state, data);
A5:     send(Actuator, cmd);
A4:     state = nextState(state, data);
        }
```

Introduction

——transforming tasks for enhanced schedulability

- The event-based semantics provides a foundation to automatically tune a real-time system
  - A compiler decomposition technique can be used to automatically decompose A4
  - A task transformation algorithm can relocate code to tolerate single-period overloads

Introduction
——transforming tasks for enhanced schedulability

- The task transformation technique is developed to support control-domain programs under rate-monotonic scheduling.

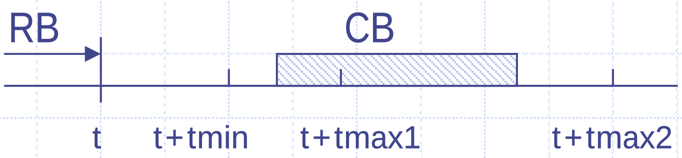
- The framework consists:
  - *An algorithm*, to find unschedulable tasks, and determine the amount that they must be transformed.
  - *A program slicer*, to decomposes a task and isolates the component that can have its deadline postponed.
  - *An online, dynamic adaptation* to modify the rate-monotonic scheduler, to enforce precedence constrains between task iterations. (adaptation priority exchange)

Overview of TCEL

- sporadic program :

do
    ⟨reference block⟩
[**start after** t_{min}] [**start before** t_{max1}]
[**finish within** t_{max2}]
    ⟨constraint block⟩

The 'do' construct induces the following timing constrains:

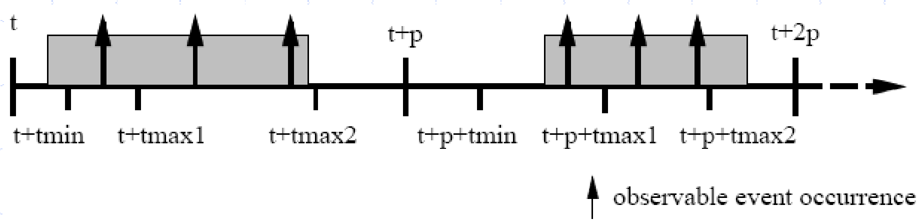


- **start after** t_{min}: There is a minimum delay of t_{min} between the last event executed in the RB, and the first event executed in the CB.
- **start before** t_{max1}: There is a maximum delay of t_{max1} between the last event executed in the RB, and the first event executed in the CB.
- **finish within** t_{max2}: There is a maximum delay of t_{max2} between the last event executed in the RB, and the last event executed in the CB.

Overview of TCEL

- periodic program

> **every** p [**while** ⟨condition⟩]
> [**start after** t_{min}] [**start before** t_{max1}]
> [**finish within** t_{max2}]
> ⟨constraint block⟩



- **start after** t_{min}: The first event executed in the CB occurs after $t + ip + t_{min}$.
- **start before** t_{max1}: The first event executed in the CB occurs before $t + ip + t_{max1}$.
- **finish within** t_{max2}: The last event executed in the CB occurs before $t + ip + t_{max2}$.

Scheduling with Compiler Transformations

- To motivate the transformation, the paper gave an example set of GN&C tasks (guidance, navigation and control), which is shown to be unschedulable with Rate-Monotonic scheduler.

Scheduling with Compiler Transformations
--characterization of control software

- One major property: control algorithms are executed repetitively with fixed periods
- During each period:
  - the physical world measurement data is sampled,
  - then, actuator commands are computed,
  - meanwhile, a set of states is updated,
- Dynamic behavior of GN&C can be expressed:

$$O_k = g(X_k, I_k) \quad (1)$$

$$X_{k+1} = h(X_k, I_k) \quad (2)$$

I_k: input of the kth period O_k: output of the kth period X_k: current state of the kth period

Scheduling with Compiler Transformations

--characterization of control software

- One possible ordering of Eq1 and 2:
  - Common computational part is factored out
  - $O_k = g(X_k, I_k) \longrightarrow$ Com;OG
  - $X_{k+1} = h(X_k, I_k) \longrightarrow$ Com;ST

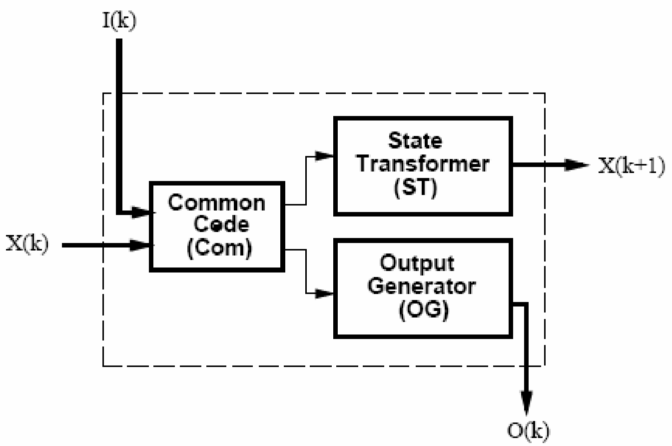


Figure 4: Task Decomposition in the k^{th} Period.

•Inter-task precedence is represented by the arrows

•Intra-task precedence :

$$(1)\ \ \texttt{Com}(\texttt{k}); \texttt{ST}(\texttt{k}) \prec \texttt{Com}(\texttt{k}+1); \texttt{ST}(\texttt{k}+1)$$
$$(2)\ \ \texttt{Com}(\texttt{k}); \texttt{ST}(\texttt{k}) \prec \texttt{Com}(\texttt{k}+1); \texttt{OG}(\texttt{k}+1)$$

Scheduling with Compiler Transformations
--Rate-Monotonic Schedulability Analysis

- A set of tasks $\tau_1, \tau_2, \ldots$
- $\tau_i(T_i, C_i)$, $T_1 < T_2 < T_3 \ldots$
- *scheduling points* are those points which are multiples of the periods of the tasks.

- To determine if task τ_k can meet its deadline under the worst case, we need to check those *scheduling points* in the interval $[0, T_k]$

$$\sum_{i=1}^{k} \frac{C_i \lceil \frac{t}{T_i} \rceil}{t} \leq 1$$

Scheduling with Compiler Transformations

--Rate-monotonic Schedulability Analysis

Example 1: Consider the case of three periodic tasks, where $U_i = C_i/T_i$.

$$\begin{aligned}
&\text{Task}(\tau_1) : C_1 = 4.0; T_1 = 10; U_1 = 0.4 \\
&\text{Task}(\tau_2) : C_2 = 4.0; T_2 = 16; U_2 = 0.25 \\
&\text{Task}(\tau_3) : C_3 = 6.41; T_3 = 25; U_3 = 0.2612
\end{aligned}$$

- τ_1 and τ_2 are schedulable, because $U_1 + U_2 < n(2^{1/n} - 1) = 2(2^{1/2} - 1) = 0.83$
- But the entire task set is not schedulable.

scheduling points within [0,T3]:

$$\begin{aligned}
&C_1 + C_2 + C_3 \le T_1 && (4 + 4 + 6.41 > 10) \\
&2C_1 + C_2 + C_3 \le T_2 && (8 + 4 + 6.41 > 16) \\
&2C_1 + 2C_2 + C_3 \le 2T_1 && (8 + 8 + 6.41 > 20) \\
&3C_1 + 2C_2 + C_3 \le T_3 && (12 + 8 + 6.41 > 25)
\end{aligned}$$

let some of τ_3's code 'slide' into the next period, to achieve schedulability.

This is called deadline postponement.

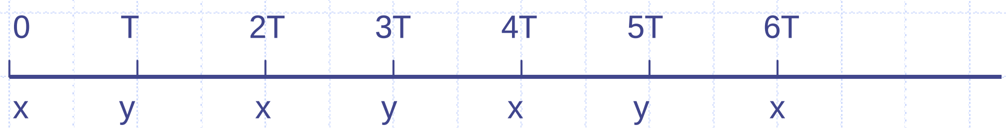
Scheduling with Compiler Transformations
--Task Transformation Algorithm

- The application of deadline postponement can be described :

Step 1 Task τ is duplicated into two tasks τ_x and τ_y.

Step 2 Both τ_x and τ_y are given $2T$ as their period, where T is τ's original period.

Step 3 τ_x is initiated at times $0, 2T, \ldots$, while τ_y is initiated at times $T, 3T, \ldots$.



- Some observable events may miss their deadlines.
  - Use a compiler-driven task decomposition technique
- How to preserve the original precedence?
  - An online, dynamic adaptation

Scheduling with Compiler Transformations
--Task Transformation Algorithm

- Task decomposition. We use the task set in Exp 1.
- Decompose τ_3's code into two parts: τ_{3a} and τ_{3b}
  1. Code that computes the output command --- τ_{3a} , correspond to 'Com, OG'
  2. Code that computes the state update --- τ_{3b} , correspond to 'ST'

```
     every 25ms
     {
L1:   receive(Sensor, data);          [0.2ms,0.5ms]
L2:   if (!null(data))                [0.05ms,0.06ms]
      {
L3:     t1 = F1(state);               [0.8ms,1.05ms]
L4:     t2 = F2(state);               [0.9ms,1.35ms]
L5:     t3 = F3(data);                [0.9ms,1.35ms]
L6:     t4 = F4(data);                [0.9ms,1.35ms]
L7:     cmd = t1 * ( t3 + t4 );       [0.09ms,0.1ms]
L8:     send(Actuator, cmd);          [0.2ms,0.5ms]
L9:     state = t1 * ( t2 + t3 );     [0.11ms,0.15ms]
      }
L10: }
```

Figure 5: TCEL Program for Task τ_3.

```
/* Subtask τ3a */
every 25ms
  {
  receive(Sensor, data);        [0.2ms,0.5ms]
  c = !null(data);              [0.05ms,0.06ms]
  if (c)                        [0.01ms,0.02ms]
    {
    t1 = F1(state);             [0.8ms,1.05ms]
    t3 = F3(data);              [0.9ms,1.35ms]
    t4 = F4(data);              [0.9ms,1.35ms]
    cmd = t1 * ( t3 + t4 );     [0.09,0.1ms]
    send(Actuator, cmd);        [0.2ms,0.5ms]
    }
  }
/* Subtask τ3b */
every 25ms
  {
  if (c)                        [0.01ms,0.02ms]
    {
    t2 = F2(state);             [0.9ms,1.35ms]
    state = t1 * ( t2 + t3 );   [0.11ms,0.15ms]
    }
  }
```

Figure 6: Two Decomposed Subtasks.

Scheduling with Compiler Transformations
--Task Transformation Algorithm

- Subtask τ_{3b} consists of only local computations, we can subject it to deadline postponement,
  - Two duplicated task: τ_{3b1} , τ_{3b2}
  - With period : $T_{3b1} = T_{3b2} = 2T_3$
  - τ_{3b2} is initiated after a delay of T_3 from the initiation of τ_{3b1}

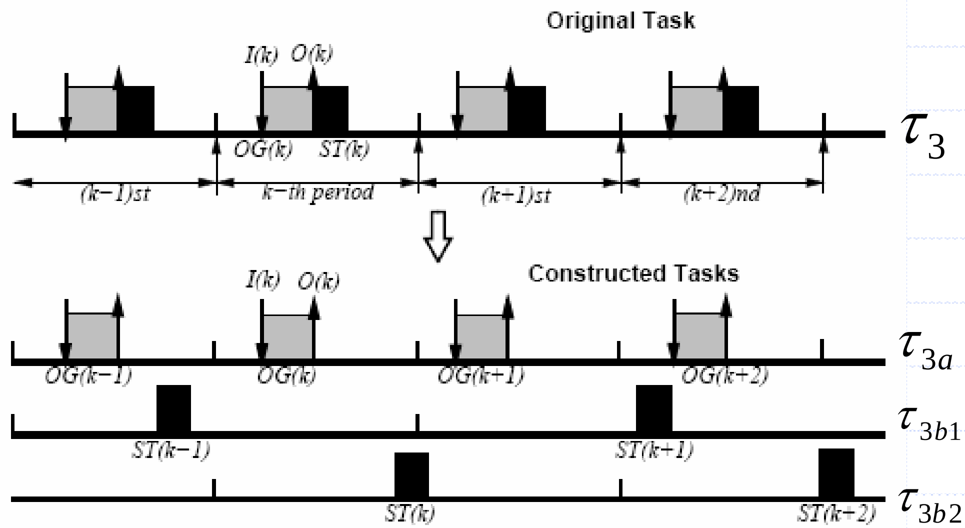


Figure 7: Scheduling of Newly Constructed Tasks.

This transformation is unsafe, unless we ensure that the precedence constraints between the tasks are maintained.

Scheduling with Compiler Transformations

--Task Transformation Algorithm

- Assume the original precedence is maintained.
- Consider the schedulability of task set $\{\tau_1, \tau_2, \tau_{3a}, \tau_{3b1}, \tau_{3b2}\}$
- For the sake of schedulability analysis, the paper coalesces τ_{3b1} and τ_{3b2} into τ_{3B}.($T_{3B} = 2T_3$ and $C_{3B} = C_{3b1} + C_{3b2}$)

$$3C_1 + 2C_2 + C_{3a} \leq T_3$$
$$(12 + 8 + 4.93 < 25)$$
$$5C_1 + 3C_2 + 2C_{3a} + C_{3B} \leq 3T_2$$
$$(20 + 12 + 9.86 + 3.04 < 48)$$

- as long as the precedence constrains are maintained, the above transformation guarantees that observable operations meet their deadlines.

Scheduling with Compiler Transformations

--Modifying the scheduler: Priority Exchange

- Scheduler: rate-monotonic scheduler
- The precedence constraints of $\{\tau_a, \tau_{b1}, \tau_{b2}\}$:

$$
\begin{array}{llll}
\text{(C1)} & \tau_{b1}^{k} \prec \tau_{b2}^{k} & \text{and} & \text{(C2)} \quad \tau_{b2}^{k} \prec \tau_{b1}^{k+1} \\
\text{(C3)} & \tau_{a}^{2k} \prec \tau_{b1}^{k} & \text{and} & \text{(C4)} \quad \tau_{a}^{2k+1} \prec \tau_{b2}^{k} \\
\text{(C5)} & \tau_{b1}^{k} \prec \tau_{a}^{2k+1} & \text{and} & \text{(C6)} \quad \tau_{b2}^{k} \prec \tau_{a}^{2(k+1)}
\end{array}
$$

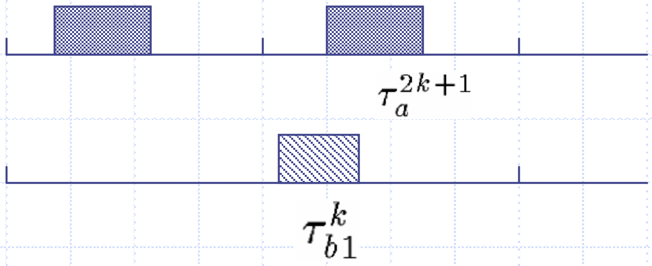
- This scheduler can keep the constraints C1 and C2 (give the two task same priority); also can keep C3 and C4.
- But this scheduler cannot guarantee C5 and C6.
- The paper introduced a dynamic modification for the scheduler called *priority exchange*.

Scheduling with Compiler Transformations

--Modifying the scheduler: Priority Exchange

- Priority exchange :
  - p_a and p_{b1} denote the priority of τ_a and τ_{b1} ($p_a > p_{b1}$)

- When a period of τ_a starts in the middle of T_{b1}, and if τ_{b1} has not yet finished its execution, then τ_{b1} exchanges its priority with τ_a. Also, a count-down timer gets set to C_{b1}.
- The timer is only decremented (1) if it has been set, *and* (2) if τ_{b1} or τ_a are *running* with priority p_a. That is, if either τ_{b1} or τ_a get preempted by a higher priority task, the timer is temporarily stopped.
- If τ_{b1} finishes *before* the timer expires, then τ_a is restored to its original priority p_a.



Automatic Task Decomposition by program Slicing

- Idea of task decomposition:
  - Accept a task, then generate its two code components ($\tau_3 \rightarrow \tau_{3a} + \tau_{3b}$)
  - One component contains observable events (τ_{3a}); the other includes the next-state update (τ_{3b}).
- Program slicing:
  - Assumption: function calls are inlined; loops are unrolled; the intermediate code of programs is translated into static single assignment form.
  - Computation of slices is based on data dependence and control dependence. We can use program dependence graph.

Automatic Task Decomposition by program Slicing

- Definition:
  - A slice of program P consists of P's statements and control predicates that may affect the value of v at point p. we call a pair <p, v> a *slicing criterion*, and denote its associated slice by P/<p, v>.
  - Example:

    the following fragment is the slice $P_{control} / < eot, state >$

    where *eot* is a pseudo-location at the end of the loop body.

```
      every 25ms
        {
L1:     receive(Sensor, data);
L2:     if (!null(data))
          {
L3:       t1 = F1(state);
L4:       t2 = F2(state);
L5:       t3 = F3(data);
L9:       state = t1 * ( t2 + t3 );
          }
        }
```

Automatic Task Decomposition by program Slicing

- Definition of program dependence graph G=(V, E):
  - The vertexes V represent the task's operations. In addition there is a distinguished vertex 'entry' , which represents the root of the task.
  - The edges E are of two sorts:

$n_1 \xrightarrow{c} n_2$
- between entry and vertex that is not nested within any loop or conditional
- between control predicate and vertex that is immediately nested within the loop or conditional

$n_1 \xrightarrow{d} n_2$
- loop independent
- loop carried

Automatic Task Decomposition by program Slicing

```
      every 25ms
        {
L1:     receive(Sensor, data);
L2:     if (!null(data))
          {
L3:       t1 = F1(state);
L4:       t2 = F2(state);
L5:       t3 = F3(data);
L6:       t4 = F4(data);
L7:       cmd = t1 * ( t3 + t4 );
L8:       send(Actuator, cmd);
L9:       state = t1 * ( t2 + t3 );
          }
L10:    }
```

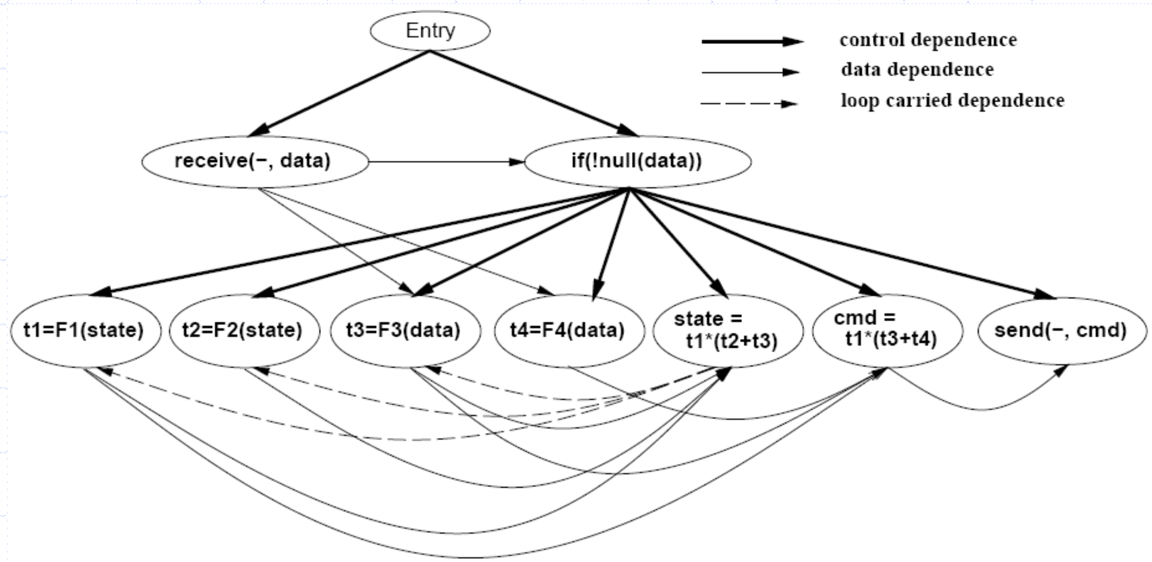


Figure 9: Program Dependence Graph.

Automatic Task Decomposition by program Slicing

- A simple method to compute the slice *P/<p, v>* :

  (the program point p corresponds to a vertex of the graph.)
  - Compute slicing criterion.
  - Compute the slice by a backward traversal of the graph

- The most important part of program slicing is to pick the right slicing criteria so that the resulting slices of a task 'cover' all behaviors of the original task.

Automatic Task Decomposition by program Slicing

- we use the two following sets of slicing criteria
  1. $C_o(\tau)$ includes all slicing criteria <o, var(o)> where *o* is an observable operation which occurs in task $\tau's$ code, and var(o) is a variable appearing in *o*.
  2. $C_s(\tau)$ includes slicing criteria <eot, s> where *s* is a state variable in the task.

Automatic Task Decomposition by program Slicing

- This decomposition is safe , because the two sets of slices $C_0(\tau)$ and $C_s(\tau)$ can preserve the task's original behavior:

variables:

- variables that affect observable operations (by data / control dependence)
- variables that do not affect (can be deleted, because they do not change the original observable behaviors)

Automatic Task Decomposition by program Slicing

- Using the two criterion sets, the task decomposition algorithm is given below:

Algorithm 4.2 *Decompose task τ into τ_a and τ_b:*

Step 1 Compute $C_o(\tau)$ and slice task τ with respect to $C_o(\tau)$. Then the generated slice $\tau/C_o(\tau)$ becomes τ_a.

Step 2 Compute $C_s(\tau)$ and slice task τ with respect and $C_s(\tau)$.

Step 3 Delete from $\tau/C_s(\tau)$ non-conditional statements common to both of the slices. The remaining code becomes τ_b.

Conclusion

- The paper presented
  - A new real time programming language, TCEL
  - A compilation technique which automates task tuning operations for enhanced schedulability