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:

\begin{center}
\textbf{do}
\begin{itemize}
  \item \textbf{(reference block)}
  \item [\textbf{start after} \(t_{\text{min}}\) \textbf{]} \textbf{[start before} \(t_{\text{max1}}\) \textbf{]}
  \item \textbf{[finish within} \(t_{\text{max2}}\) \textbf{]}
  \item \textbf{(constraint block)}
\end{itemize}
\end{center}

The ‘do’ construct induces the following timing constrains:

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

periodic program

- **start after** $t_{\text{min}}$: The first event executed in the CB occurs after $t + ip + t_{\text{min}}$.
- **start before** $t_{\text{max}1}$: The first event executed in the CB occurs before $t + ip + t_{\text{max}1}$.
- **finish within** $t_{\text{max}2}$: The last event executed in the CB occurs before $t + ip + t_{\text{max}2}$.
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.
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:

\[ O_k = g(X_k, I_k) \]  \hspace{0.5cm} (1)

\[ X_{k+1} = h(X_k, I_k) \]  \hspace{0.5cm} (2)

\( I_k \): input of the \( k \)th period  \( O_k \): output of the \( k \)th period  \( X_k \): current state of the \( k \)th period
Scheduling with Compiler Transformations
--characterization of control software

One possible ordering of Eq1 and 2:

- Common computational part is factored out
  - $O_k = g(X_k, I_k)$ $\rightarrow$ Com; OG
  - $X_{k+1} = h(X_k, I_k)$ $\rightarrow$ Com; ST

- Inter-task precedence is represented by the arrows
- Intra-task precedence:
  1. $\text{Com}(k); \text{ST}(k) \prec \text{Com}(k+1); \text{ST}(k+1)$
  2. $\text{Com}(k); \text{ST}(k) \prec \text{Com}(k+1); \text{OG}(k+1)$

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

**\begin{array}{cccccc}
0 & T_1 & T_2 & 2T_1 & 2T_2 \\
\end{array}**

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

\[
\sum_{i=1}^{k} \frac{C_i \left\lfloor \frac{t}{T_i} \right\rfloor}{t} \leq 1
\]
Scheduling with Compiler Transformations
--Rate-monotonic Schedulability Analysis

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

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

• $\tau_1$ and $\tau_2$ 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]:

\[
\begin{align*}
C_1 + C_2 + C_3 &\leq T_1 & (4 + 4 + 6.41 > 10) \\
2C_1 + C_2 + C_3 &\leq T_2 & (8 + 4 + 6.41 > 16) \\
2C_1 + 2C_2 + C_3 &\leq 2T_1 & (8 + 8 + 6.41 > 20) \\
3C_1 + 2C_2 + C_3 &\leq T_3 & (12 + 8 + 6.41 > 25)
\end{align*}
\]

let some of $\tau_3$'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 $\tau$ is duplicated into two tasks $\tau_x$ and $\tau_y$.

**Step 2** Both $\tau_x$ and $\tau_y$ are given $2T$ as their period, where $T$ is $\tau$'s original period.

**Step 3** $\tau_x$ is initiated at times $0, 2T, \ldots$, while $\tau_y$ 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 $\tau_3$'s code into two parts: $\tau_{3a}$ and $\tau_{3b}$

1. Code that computes the output command --- $\tau_{3a}$, correspond to ‘Com, OG’
2. Code that computes the state update --- $\tau_{3b}$, correspond to ‘ST’

```c
/* Subtask $\tau_{3a}$ */
every 25ms
{
    receive(Sensor, data);   [0.2ms,0.5ms]
    if (!null(data))
    {
        L1:  t1 = F1(state);        [0.8ms,1.05ms]
        L2:  t2 = F2(state);        [0.9ms,1.35ms]
        L3:  t3 = F3(data);        [0.9ms,1.35ms]
        L4:  t4 = F4(data);        [0.9ms,1.35ms]
        L5:  cmd = t1 * ( t3 + t4 );  [0.09ms,0.1ms]
        L6:  send(Actuator, cmd);   [0.2ms,0.5ms]
        L7:  state = t1 * ( t2 + t3 );  [0.11ms,0.15ms]
    }
    L8:  }
    L9:  }
    L10: }
```

Figure 5: TCEL Program for Task $\tau_3$.

```c
/* Subtask $\tau_{3b}$ */
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 $\tau_{3b}$ consists of only local computations, we can subject it to deadline postponement,

- Two duplicated task: $\tau_{3b1}, \tau_{3b2}$
- With period: $T_{3b1} = T_{3b2} = 2T_3$
- $\tau_{3b2}$ is initiated after a delay of $T_3$ from the initiation of $\tau_{3b1}$

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

Figure 7: Scheduling of Newly Constructed Tasks.
Scheduling with Compiler Transformations
--Task Transformation Algorithm

- Assume the original precedence is maintained.
- Consider the schedulability of task set \{τ₁, τ₂, τ₃a, τ₃b₁, τ₃b₂\}
- For the sake of schedulability analysis, the paper coalesces τ₃b₁ and τ₃b₂ into τ₃B. (T₃B = 2T₃ and C₃B = C₃b₁ + C₃b₂)

\[
3C₁ + 2C₂ + C₃a \leq T₃ \\
(12 + 8 + 4.93 < 25)
\]
\[
5C₁ + 3C₂ + 2C₃a + C₃B \leq 3T₂ \\
(20 + 12 + 9.86 + 3.04 < 48)
\]

- As long as the precedence constraints 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 \( \{\tau_a, \tau_{b1}, \tau_{b2}\} \):

(C1) \( \tau_{b1}^k < \tau_{b2}^k \) and (C2) \( \tau_{b2}^k < \tau_{b1}^{k+1} \)

(C3) \( \tau_a^{2k} < \tau_{b1}^k \) and (C4) \( \tau_a^{2k+1} < \tau_{b2}^k \)

(C5) \( \tau_{b1}^k < \tau_a^{2k+1} \) and (C6) \( \tau_{b2}^k < \tau_a^{2(k+1)} \)

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 \( p_{b_1} \) denote the priority of \( \tau_a \) and \( \tau_{b_1} \) (\( p_a > p_{b_1} \))

- When a period of \( \tau_a \) starts in the middle of \( T_{b_1} \), and if \( \tau_{b_1} \) has not yet finished its execution, then \( \tau_{b_1} \) exchanges its priority with \( \tau_a \). Also, a countdown timer gets set to \( C_{b_1} \).

- The timer is only decremented (1) if it has been set, and (2) if \( \tau_{b_1} \) or \( \tau_a \) are running with priority \( p_a \). That is, if either \( \tau_{b_1} \) or \( \tau_a \) get preempted by a higher priority task, the timer is temporarily stopped.

- If \( \tau_{b_1} \) finishes before the timer expires, then \( \tau_a \) 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 ($\tau_3a$); the other includes the next-state update ($\tau_3b$).

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 \( P/\langle p, v \rangle \).

- Example:
  
  the following fragment is the slice \( P_{\text{control}}/\langle \text{eot}, \text{state} \rangle \)

  where \( \text{eot} \) is a pseudo-location at the end of the loop body.

```c
  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:
  - between entry and vertex that is not nested within any loop or conditional
    \[ n_1 \xrightarrow{c} n_2 \]
  - 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 \( P/\langle p, v \rangle \):

(\textit{the program point } p \textit{ 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 $<o, \text{var}(o)>$ where $o$ is an observable operation which occurs in task $\tau'$s code, and $\text{var}(o)$ is a variable appearing in $o$.

2. $C_s(\tau)$ includes slicing criteria $<\text{eot}, s>$ 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 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 $\tau$ into $\tau_a$ and $\tau_b$:

1. **Step 1** Compute $C_o(\tau)$ and slice task $\tau$ with respect to $C_o(\tau)$. Then the generated slice $\tau/C_o(\tau)$ becomes $\tau_a$.
2. **Step 2** Compute $C_s(\tau)$ and slice task $\tau$ with respect and $C_s(\tau)$.
3. **Step 3** Delete from $\tau/C_s(\tau)$ non-conditional statements common to both of the slices. The remaining code becomes $\tau_b$. 
Conclusion

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