In the forthcoming *Principles of Real-Time Systems.* (Sang Son ed.), Prentice Hall, 1994.

Compiler Support for Real-Time Programs

Richard Gerber and Seongsoo Hong
Department of Computer Science
University of Maryland
College Park, MD 20742
(301) 405-2710
`rich@cs.umd.edu` `sshong@cs.umd.edu`

University of Maryland Technical Report
UMD CS-TR-3217, UMIACS-TR-94-15

January 1994

Abstract

We present a compiler-based approach to automatically assist in constructing real-time systems. In this approach, source programs are written in TCEL (or Time Constrained Event Language) which possesses high-level timing constructs, and whose semantics characterizes time-constrained relationships between observable events. A TCEL program infers only those timing constraints necessary to achieve real-time correctness, without over-constraining the system. We exploit this looser semantics to help transform programs to automatically achieve schedulability. In this article we present two such transformations. The first is trace-scheduling, which we use to achieve consistency between a program's worst-case execution time and its real-time requirements. The second is program-slicing, which we use to automatically tune application programs driven by rate-monotonic scheduling.

1 Introduction

One of the primary challenges of building a real-time system lies in balancing its functional requirements against its temporal requirements. Functional requirements define valid translations from inputs into outputs. As such they are realized by a set of programs, which *consume* CPU time.

Temporal requirements, on the other hand, place upper and lower bounds between *occurrences of events* [3, 14]. An example is *the robot arm must receive a next-position update every 10 ms.* Such a constraint arises from the system's requirements, or from a detailed analysis of the application environment. Thus temporal requirements implicitly *limit* the time that can be provided by the system's resources. When the balance between the functional and temporal constraints is not achieved, the result is often a costly and arduous process of system tuning. This typically involves multiple phases of instrumentation and hand-optimization.

In this article we present an automated methodology to assist programmers in this process. Our approach consists of two interrelated factors: a programming language and compiler transformations. The real-time programming language is called TCEL (Time-Constrained Event Language), which contains high-level timing constructs, and whose semantics is based on time-constrained relationships between observable events. As the only timing constraints are imposed by observable events, the unobservable code can be transformed to automatically assist in the low level tuning process. As we show in this article, it is precisely the TCEL semantics which makes the compiler transformations possible.

The TCEL Language. TCEL contains constructs quite similar to those developed in other experimental languages [13, 16, 18, 20, 23, 31]. In these approaches, however, timing constraints are established between *blocks of code*. The TCEL semantics, on the other hand, establishes constraints between the *observable events* within the code.

For example, consider a construct such as "**every** 10ms **do** B," where the block of code "B" is executed once every 10 milliseconds. In a code-based semantics, *all* of the code in B must fit properly within each 10 ms time-frame. In the TCEL event-based semantics, *only the observable events* in B must fit properly within the time-frame. This looser semantics yields two immediate benefits. First, the decoupling of timing constraints from code blocks enables a more straightforward implementation of an event-based specification. But more importantly, the unobservable code can be moved to automatically tune the program to its hardware environment.

In the sequel we consider all "**send**" and "**receive**" operations to be observable. As an example, the following is a fragment of a periodic TCEL program. During each period, sensor data is read in, a new system state is updated, an actuator command is determined, after which it is sent to the actuator.

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

The system's only observable events are triggered instantaneously during the executions of the "**send**" and "**receive**" operations. The "**every**" statement establishes timing constraints *only* between these two operations. On the other hand, the local statements A3 and A4 are simply constrained by the program's natural control and data dependences.

Under a code-based semantics the program is interpreted in a different way; that is, the statements A2-A5 must be executed within a single frame. This interpretation is in fact *much* stronger than the requirements mandate, and indeed, may result in an unnecessary fault. For example, if the system experiences a transient overload caused by higher-priority tasks, the program may not meet its deadline.

In this case there are obvious remedies, which *would have to be performed by hand.* For example, part or all of the next-state update in A3 could be relocated beyond A5. Then, in the case of transient overload, this operation could be postponed beyond the end of the period. However, the necessary corrections would include manually decomposing A4, moving part of it, and adding necessary hooks for the scheduler to postpone a deadline. The actual changes would heavily depend on the particular characteristics of the computer, and thus, the very reason for using high-level timing constructs would be defeated.

The event-based semantics provides a solid foundation for a compiler to automatically tune the system. Returning to our example, a transformation can be used to automatically decompose A3. Yet another transformation can relocate as much (or as little) code as is necessary to tolerate single-period overloads. In performing these transformations, the TCEL compiler uses the observable events as "semantic markers," which establish boundaries of code decomposition, and constrain the places where code can be moved.

Contents of this Chapter. This article summarizes our recent results in program transformations for real-time applications (for a more comprehensive treatment, see [11] and [9]). In Section 2 we present an overview of the TCEL language, stressing mainly the event-based semantics.

Then, in Section 3 we show how we use code-motion optimizations to resolve conflicts within single tasks. These conflicts can arise when tasks have nested constraints; e.g., when deadlines are tighter than periods, or when there are inserted delay statements. Since these timing constraints may conflict with the task's execution time, it may appear to be inherently unschedulable. Hence the objective is to automatically achieve "internal" consistency between real-time requirements and elapsed execution time. Our approach is to use instruction-scheduling techniques [7], with which our compiler moves code from blocks constrained by tight deadlines into blocks with sufficient slack.

In Section 4 we address a more aggressive goal – inter-task transformations for schedulability. To accomplish this we apply the technique of *program slicing* [24, 29, 30], in which a single task is split into multiple threads, based on the application's real-time requirements. This method is particularly amenable to control-domain programs running under rate-monotonic scheduling [21]. Using this method, our algorithm converts an unschedulable task set into a schedulable one, by isolating the time-critical threads from each control task, and ensuring that they can be scheduled. In particular, this technique is a *safe*, *automatic* way to apply deadline postponement [26] to the unobservable threads.

We conclude in Section 5 with a discussion on the practical implications of our work.

2 Overview of TCEL

In this section we present two of TCEL's constructs to denote timing constraints within a program. Both constructs are syntactic descendents of the *temporal scope*, introduced in [18]. However, as we have stated, our semantics is quite different, in that it relies on constrained relationships between observable events.

We use the "**do**" construct to denote a sporadic program with relative timing constraints:

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

The reference block (RB) and the constraint block (CB) are simply C statements, or alternatively, timing constructs themselves. The "**do**" construct induces the following timing constraints:

- **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.

Since either block may contain conditionals, depending on the program's state there may be several such events executed either "first" or "last." For example, consider the fragment from a typical flow graph in Figure 1.

Depending on the path taken, the last event executed in the reference block may be either E1 or E2. Similarly, the first event in the constraint block will be E3 or E4, while the last event will be either E4 or E5. To denote such possibilities, we introduce two mappings *FIRST* and *LAST* from code blocks to sets

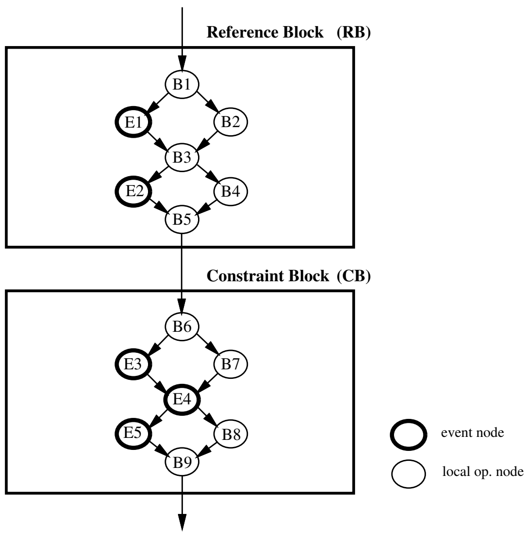

Figure 1: Typical Flow Graph.

of events. That is, $LAST(RB) = \{E1, E2\}$, $FIRST(CB) = \{E3, E4\}$ and $LAST(CB) = \{E4, E5\}$. Thus, the "**do**" construct introduces two potential constraints between an executed event from $LAST$(RB) and another from $FIRST$(CB), as well as one constraint between two events from $LAST$(RB) and $LAST$(CB) each.

The second real-time construct denotes a statement with cyclic behavior of a positive periodicity:

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

As long as the "**while**" condition is true, the observable events in the constraint block execute every p time units. Akin to an untimed **while**-loop, when the condition evaluates to false the statement terminates. In its real-time behavior, the interpretation of the "**every**" construct is similar to that of "**do**." For example, assume that the statement is first scheduled at time t, and that the "**while**" condition is true for periods 0 through i. As depicted in Figure 2, the following constraints on events are induced for period i:

- **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 $t + ip + t_{max1}$.

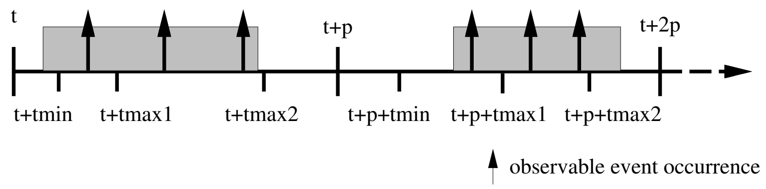

Figure 2: Behavior of Periodic Timing Construct.

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

As we have stated, timing constraints may be arbitrarily nested. For example, consider the two-arm robot control program in Figure 3, which monitors a conveyer belt and gives commands to the robot's arms. The specification is as follows:

(1) Every 10 ms, a position-sensor sends a message to the controller.

(2) Each message contains the dimensions of an object currently approaching the robot. If no object is approaching, the message is tagged as "null."

(3) To achieve steady-state, the controller delays at least 1.5 ms after receiving the message.

(4) If an object was detected, new commands are sent to arm1 and arm2.

(5) Both commands must be sent within 4.0 ms of receiving the sensor's message, and within 8.0 ms of the beginning of the period.

3 Transformation 1: Consistent Task Synthesis

While an event-based semantics makes sense at the source-program level, most real-time schedulers only accept timing constraints on the start and finish times of *tasks*. Thus, the role of the compiler is to transform event-driven source programs into constrained blocks of code. The challenge is to achieve a task set which is *feasible*, i.e., whose tasks have execution times consistent with the timing constraints.

This is done in the following three steps. First, a timing construct is decomposed into several *sections*, denoted by its control flow structure (Section 3.1). Next, code-based timing constraints are derived from the construct's event-based timing constraints (Section 3.2), and checked for their consistency with the execution time. Finally, code-scheduling transformations are used to reduce the worst-case execution time of the infeasible sections (Section 3.3).

3.1 Section Generation

A timing construct is divided into five code sections, as portrayed in Figure 4. As can be seen, the reference block is decomposed into three sub-blocks. The unobservable code before the first observable statement becomes an interface section (S1).

```
every 10 ms finish within 8 ms
  do
    {
    receive(Sensor, &dim);
    msg_cnt++;
    }
  start after 1.5 ms finish within 4 ms
    {
    if (!null(dim))
      {
      z1 = convert(dim, loc1);
      z2 = convert(dim, loc2);
      send(arm1, z1);
      send(arm2, z2);
      }
    }
```

Figure 3: Source Code for Robot Controller.

The code containing the observable statements becomes the reference section (S2). The unobservable code after the observable statements becomes the first part of the delay section (S3). Consequently, the topmost unobservable code of the constraint block becomes the second part of S3, and so on.

Recall the discussion of the $FIRST$ and $LAST$ functions in Section 2. Since a code block may contain complicated control structures, we require a convenient means of defining the boundaries of S2 and S4 – the sections that contain observable events. We accomplish this by inserting "markers" in the flow graph, which consume no time and are not visible. The following marker definitions guarantee that there are unique boundaries into and out of the sections containing observable events.

- begin_S2: This marker is inserted directly after the unobservable instruction most closely dominating $LAST(\mathrm{RB}) \cup \{exit(\mathrm{RB})\}$.
- end_S2: This marker is inserted directly before the unobservable instruction most closely post-dominating $LAST(\mathrm{RB}) \cup \{entry(\mathrm{RB})\}$.
- begin_S4: This marker is inserted directly after the unobservable instruction most closely dominating $FIRST(\mathrm{CB}) \cup \{exit(\mathrm{CB})\}$.
- end_S4: This marker is inserted directly before the unobservable instruction most closely post-dominating $LAST(\mathrm{CB}) \cup \{entry(\mathrm{CB})\}$.

For example, consider the constraint block in Figure 4. The unobservable node B9 post-dominates $LAST(\mathrm{CB})$ and the entry node. Thus, its logical place

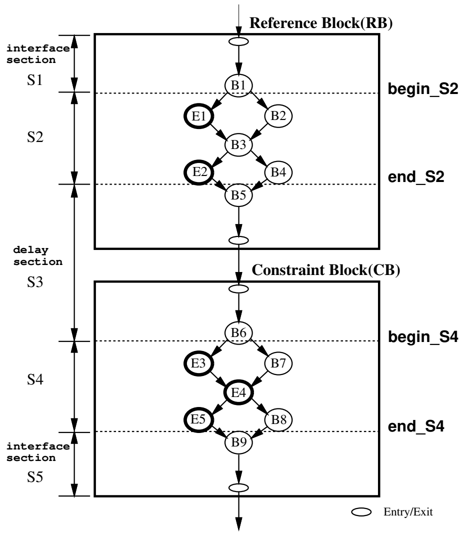

Figure 4: The Flow Graph of a Timing Construct and its Section Division.

is in the interface section S5, which is not subject to the construct's timing constraints. Hence the need for the marker end_S4, which is the unique exit point for the constrained section S4.

Now, let the variable S2.start correspond to the actual time that the marker begin_S2 is "executed" (that is, the dispatch time of section S2), and let S2.finish correspond to the time that the section ends. Similarly, let S4.start and S4.finish represent the start and finish times of section S4. Using these variables we can represent the section decomposition of a TCEL construct in a manner similar to that found in the Flex language [15].

Recall the robot controller program from Figure 3. Figure 5 illustrates its constituent sections, where the bracketed numbers are the maximum execution times for each instruction on the given CPU. These times are generated by a timing analysis tool, such as those found in [10, 5, 25, 27, 32]. The constraint-expression for S6 corresponds to the program's outer, periodic loop.

3.2 Deriving Code-Based Timing Constraints

As seen in Figure 5, the code-based timing constraints can be expressed as conjunctions of linear inequalities between start-times and finish-times of different sections. However, note the difference between the code-based constraints and the TCEL source-level constraints: In Figure 3 the "**finish within**" deadline is 4ms, while in Figure 5 it is tightened to 3.6ms. There is good reason for this – the new code-based timing constraints must be strong enough to guarantee the original semantics of the event-based constraints. That is, they must take into account the program's execution-time characteristics. In general, consider the TCEL construct such as

```
S6: (S6.start[p]≥p×10ms,
      S6.finish[p]≤p×10ms+8ms)
S1:    { /* null */ }
S2:    { receive(Sensor, &dim); }        [0.40ms]
S3:    {
         msg_cnt++;                      [0.02ms]
         c = !null(dim);                 [0.20ms]
       }
S4:    (S4.start≥S2.finish+1.5ms,
         S4.finish≤S2.finish+3.6ms)
       {
         if (c) {                        [0.02ms]
           z1 = convert(dim, loc1);      [1.00ms]
           z2 = convert(dim, loc2);      [1.00ms]
           send(arm1, z1);               [0.40ms]
           send(arm2, z2);               [0.40ms]
         }
       }
S5:    { /* null */ }
     }
```

Figure 5: Robot Program – After Section Generation.

do RB **start after** t_{min} **start before** t_{max1} **finish within** t_{max2} CB

Obviously, the TCEL parameters are not tight enough to guarantee the correctness of the code-based constraints. For example, if we wish to maintain the "t_{max1}" requirement, it is not sufficient to simply mandate that S4 starts within a maximum delay of t_{max1} after S2 ends (though this is certainly necessary). We can see in Figure 4 that the event actually *executed* in *LAST*(S2) may be E1, while the event executed in *FIRST*(S4) may be E4. Thus the naive strategy fails to factor in the execution times of B3 and B4.

However, the event-based semantics is clear: the time between the *executed* event in *LAST*(S2) and the *executed* event in *FIRST*(S4) is at most t_{max1}. To guarantee that this occurs, we must account for *all* possible execution scenarios. Specifically, we must tighten the constraints, allowing for the maximum amount of time between an event in *LAST*(S2) and end_S2, as well as the maximum amount of execution time between begin_S4 and an event in *FIRST*(S4). We must similarly adjust t_{max2}. To do this, we make the following definitions:

- $\Delta_{\mathrm{S2}} \stackrel{\mathrm{def}}{=} max\{wt(p) \mid e \in LAST(S2),\ e \Rightarrow^p_+ \mathrm{end\_S2}\}$.
- $\Delta_{\mathrm{S4}} \stackrel{\mathrm{def}}{=} max\{wt(p) \mid e \in FIRST(S4),\ \mathrm{begin\_S4} \Rightarrow^p_+ e\}$.

where

- For nodes $n1$ and $n2$ in the flow graph, $n1 \Rightarrow^p_+ n2$ means there is a non-null path from $n1$ to $n2$.

- For a fragment of code c, $wt(c)$ is the worst-case execution time of c.

Note that Δ_{S2} and Δ_{S4} are sensitive to not only code's execution time characteristics, but also changes made to some paths between events and markers during program translation. For example, changes to paths between end_S2 and a node in $LAST$(S2) might require re-evaluation of Δ_{S2}.

Now the code-based timing constraints can be postulated as follows:

$$\begin{array}{llll}
(1) & \text{S4.start} & \geq & \text{S2.finish} + T_{min} \quad (\text{where } T_{min} = t_{min}) \\
(2) & \text{S4.start} & \leq & \text{S2.finish} + T_{max1} \quad (\text{where } T_{max1} = t_{max1} - \Delta_{S2} - \Delta_{S4}) \\
(3) & \text{S4.finish} & \leq & \text{S2.finish} + T_{max2} \quad (\text{where } T_{max2} = t_{max2} - \Delta_{S2})
\end{array}$$

These timing constraints are strong enough to guarantee the original event-based timing constraints. (By convention, if the "**start after**" constraint is omitted, we consider t_{min} to be 0. Similarly, when either the "**start before**" or "**finish within**" constraints are missing, we consider $t_{max1} = \infty$ or $t_{max2} = \infty$, respectively.) Returning to Figure 5, we can see that equation (3) indeed mandates tightening the original 4ms to 3.6ms.

Now we wish to determine when (1)-(3) can be met. That is, what do these equations infer about the program's allowable worst-case execution-time behavior? This can easily be derived if we add precedence constraints reflecting the natural flow of the program; i.e., that S4 executes after S3, which executes after S2:

$$\begin{array}{lrcl}
(4) & \text{S2.finish} + wt(\text{S3}) & \leq & \text{S4.start} \\
(5) & \text{S4.start} + wt(\text{S4}) & \leq & \text{S4.finish}
\end{array}$$

Eliminating S2.finish, S4.start and S4.finish from (1)-(5), we end up with:

$$\begin{array}{lrcl}
(a) & T_{min} & \leq & T_{max1} \\
(b) & wt(\text{S3}) & \leq & T_{max1} \\
(c) & wt(\text{S3}) + wt(\text{S4}) & \leq & T_{max2} \\
(d) & wt(\text{S4}) & \leq & T_{max2} - T_{min}
\end{array}$$

Obviously, (a) had better be true in order for the TCEL construct to make any sense. For the purposes of our algorithm we combine (b) and (c), yielding the following two constraints on execution times:

$$\begin{array}{lrcl}
(*) & wt(\text{S3}) & \leq & min\{T_{max1}, T_{max2} - wt(\text{S4})\} \\
(**) & wt(\text{S4}) & \leq & T_{max2} - T_{min}
\end{array}$$

These are the necessary and sufficient conditions to achieve feasibility, and they are summarized in Table 1. In the next subsection we discuss our code-scheduling techniques to handle the cases in which one of these conditions fails to hold.

3.3 Code Scheduling

The final step is to rearrange instructions across sections in such a way that all the sections satisfy their derived timing constraints. Such a process is similar to that of *code scheduling*, which is a well-defined problem for automatic fine-grain (instruction level) parallelization for superscalar and VLIW processors [1, 4, 7, 8,

Section	Duration Constraint ($DUR(S)$)
S3	$min\{T_{max1}, T_{max2} - wt(S4)\}$
S4	$T_{max2} - T_{min}$

Table 1: Timing Constraints of S3 and S4.

Algorithm *Code_Scheduling*(T) /* T is a timing construct */
input: the ordered set of sections $\{S1, S2, \ldots, S5\}$ in T
begin
    dur = $T_{max2} - T_{min}$;
    compute t such that $wt(t) = max\{wt(path) \mid path \text{ in S4}\}$;
    while ($wt(t) >$ dur)
        perform *Code_Scheduling* on t into S3;
        if (t is still critical) **then** exit("Unable to synthesize.");
        recompute t such that $wt(t) = max\{wt(path) \mid path \text{ in S4}\}$;
    end
    recompute T_{max1}; /* to reflect the change in Δ_{S4}. */
    if ($wt(S3) \leq T_{min}$) **then** exit("No scheduling needed for S3.");
    else dur = $\min\{T_{max1}, T_{max2} - wt(S4)\}$;
    compute t such that $wt(t) = max\{wt(path) \mid path \text{ in S3}\}$;
    while ($wt(t) >$ dur)
        perform *Code_Scheduling* on t into S1;
        if (t is still critical) **then** exit("Unable to synthesize.");
        recompute t such that $wt(t) = max\{wt(path) \mid path \text{ in S3}\}$;
    end
end

Figure 6: Code Scheduling Algorithm.

22, 28]. However, our problem context has a different goal. In what follows, we sketch a code scheduling algorithm, which moves code from sections that violate their duration constraints into those with more lenient constraints.

Code scheduling involves copying or relocating unobservable instructions[1] to new locations, while preserving the functional semantics of the original code. In doing so, we attempt to achieve the following goal:

- Satisfy $wt(Si) \leq DUR(Si)$ for $i = 3, 4$.

The algorithm is greedy, and it attempts to attain the desired consistency of a timing construct in a section-by-section manner. It inspects the sections S4 and S3 (in reverse topological order), checking if they satisfy their duration constraints.

[1] We conservatively prohibit event-generating instructions from being moved, so that the timing relationships between events are preserved.

```
S6: (S6.start[p]≥p×10ms, S6.finish[p]≤p×10ms+8ms)
    {
S1:   { /* null */ }
S2:   { receive(Sensor, &dim); }
S3:   {
        msg_cnt++;
        c = !null(dim);
        if (c) z1 = convert(dim, loc1);
      }
S4:   (S4.start≥S2.finish+1.5ms, S4.finish≤S2.finish+3.6ms)
      {
        if (c) {
          z2 = convert(dim, loc2);
          send(arm1, z1);
          send(arm2, z2);
        }
      }
S5:   { /* null */ }
    }
```

Figure 7: Robot Program – After Code Scheduling.

If S4 violates its duration constraint, the algorithm attempts to reduce its surplus execution time by moving instructions to section S3. In turn, it processes section S3, which may now contain newly moved code.

To actually perform greedy code motion, we have adapted the approach to Trace Scheduling in [7], and we use it as a component of the code scheduling algorithm (please consult [11] for more details). Instructions lying on paths that exceed their section's duration constraints are considered for code motion. We distinguish such paths as *critical traces*. Formally, the critical trace t of section S is defined as a path from *entry*(S) to *exit*(S) such that $wt(t) > DUR(\mathrm{S})$.

Figure 6 sketches the algorithm. Note that T_{max1} is recomputed after scheduling S4 and before scheduling S3. This is mandatory, since Δ_{S4} may be changed during the scheduling. Also observe that the code of S3 is moved into S1, while that of S4 is moved into S3. We disallow code from moving into S2 because it could potentially change the value of Δ_{S2}, which would, in turn, invalidate our assumptions about $DUR(\mathrm{S4})$. In order to complete the procedure in a single pass, we assume that Δ_{S2} remains constant. In reality this restriction does not seriously limit the approach: from our experience, events in the RB typically lie in straight-line code (and thus S2 contains a single instruction).

Figure 7 illustrates the code scheduling process for our robot example. The duration constraint for section S4 is violated, since its predicted worst case execution is 2.82 ms. However, S4 is allowed only 2.1 ms to execute its body.[2] Figure 7

[2] 2.1ms = S4.finish − S4.start = (S2.finish + 3.6ms) − (S2.finish + 1.5ms)

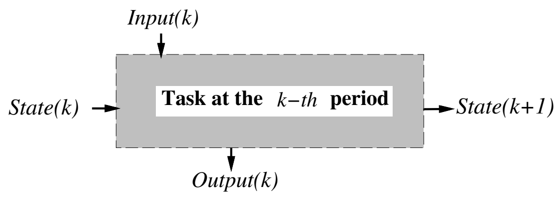

Figure 8: Task Behavior at k^{th} Period.

yields the result of code scheduling: an assignment is moved into S3, and the test guarding it is copied. Now the implementation satisfies the necessary condition for feasibility, since the body of S3 requires at most 1.82 ms. In addition to such an instant benefit, the transformation converts the possibly wasteful delay into useful computation time, since the new code in S3 can be scheduled within the delay interval between S2 and S4.

4 Transformation 2: Real-Time Schedulability

The event-based semantics of TCEL gives a clear separation between constraints based on time, and those based on data and control dependences. In this section we present another useful application of this semantics; namely, an automated tuning tool for enhancing the schedulability of (purely) periodic task sets. Thus we are concerned with a subset of TCEL, i.e., tasks written only using the **every** construct.

Our approach is as follows: whenever a task set is found unschedulable, we identify the tasks that miss their deadlines. Each of these tasks gets split into two threads – one containing its observable events, and the other containing its unobservable instructions. While the former thread must finish by the original deadline, the latter is allowed to "slide" into the next frame. Thus the transformation effectively increases the original task's deadline, while maintaining its semantics.

This approach is particularly appropriate for programs that drive guidance, navigation and control (GN&C) applications [17]. First, GN&C programs typically possess periodic behavior; thus they are amenable to fixed-priority, rate-driven scheduling. Second, these programs possess structure resembling that displayed in Figure 8: During each period, physical world measurements are sampled, a current global state is updated, actuator commands are computed, which are then sent to a set of actuators.

Since the events (i.e., inputs and outputs) are clearly identifiable here, this type of process easily lends itself to aggressive code transformations, assuming the underlying the semantics of TCEL.

4.1 Rate-Monotonic Schedulability Analysis

Rate-monotonic scheduling is well-suited for control domain applications, not only because they possess the periodic behavior, but also because efficient schedulability tests can be applied. One of these is the exact (necessary and sufficient) test presented in [19], which is based on *critical instant* analysis.

To review, a task's critical instant occurs whenever it is initiated simultaneously with all higher-priority tasks [21]. Let T_i and C_i be the period and the worst case computation time of task τ_i, respectively, and assume that the τ_i's are numbered in the increasing order of their T_i's. Since the rate-monotonic scheduling algorithm assigns a higher priority to a task with a smaller period, a task with a smaller number has a higher priority.

To determine if task τ_k can meet its deadline under the worst case phasing, it is necessary to check if there is point t in time in the interval $[0, T_k]$ (i.e. τ_k's critical interval), which satisfies the following inequality.

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

To do so, we need only check those points in the interval $[0, T_k]$ which are multiples of the periods of k tasks $\{\tau_1, \tau_2, \ldots, \tau_k\}$. They are called *scheduling points*, and become points where the left-hand-side of the above inequality achieves the local minima.

Example 4.1 Consider the case of three periodic tasks, where the source code of task τ_3 is given in Figure 9.

Task	Execution Time	Period
τ_1	$C_1 = 4.00$	$T_1 = 10$
τ_2	$C_2 = 4.00$	$T_2 = 16$
τ_3	$C_3 = 6.41$	$T_3 = 25$

We can carry out the exact schedulability test for these tasks as follows:

For τ_1:	C_1	$=$	4.00	$<$	$T_1 = 10$
For τ_2:	$C_1 + C_2$	$=$	8.00	$<$	$T_1 = 10$
For τ_3:	$C_1 + C_2 + C_3$	$=$	14.41	$>$	$T_1 = 10$
	$2C_1 + C_2 + C_3$	$=$	18.41	$>$	$T_2 = 16$
	$2C_1 + 2C_2 + C_3$	$=$	22.41	$>$	$2T_1 = 20$
	$3C_1 + 2C_2 + C_3$	$=$	26.41	$>$	$T_3 = 25$

That is, tasks τ_1 and τ_2 are schedulable, since they can both complete before their (shared) scheduling point T_1. However, the entire task set is not schedulable, because the total utilization factor exceeds 1 at all scheduling points within the critical interval of τ_3. □

4.2 Transformation Approach

A straightforward technique to achieve schedulability is to let some of τ_3's code "slide" into the next period. This can be done by postponing the deadline of τ_3 (as

```
         every 25ms
           {
L1:        receive(Sensor, &data);        [0.50ms]
L2:        if (!null(data))               [0.06ms]
             {
L3:          t1 = F1(state);              [1.05ms]
L4:          t2 = F2(state);              [1.35ms]
L5:          t3 = F3(data);               [1.35ms]
L6:          t4 = F4(data);               [1.35ms]
L7:          cmd = t1 * ( t3 + t4 );      [0.10ms]
L8:          send(Actuator, cmd);         [0.50ms]
L9:          state = t1 * ( t2 + t3 );    [0.15ms]
             }
L10:       }
```

Figure 9: TCEL Program for Task τ_3.

suggested by Sha et al. in [26]). However, since τ_3 contains critical IO operations, this technique is not always safe with respect to the event-based semantics. In this section we show how to transform the task so that the original semantics is preserved. The high-level approach is as follows:

- Given an unschedulable task τ with period T, decompose it into
  (1) τ_a, a subtask containing all of τ's observable operations.
  (2) τ_b, a subtask containing the remaining operations.
- Ensure that τ_a followed by τ_b exhibits the same functionality of the τ.
- τ_a keeps τ's original period T, so that the IO operations execute within their deadline.
- Two iterations of τ_b must execute within a period of $2T$.
- The precedence constraints between τ_a and τ_b must be maintained; i.e., the execution behavior will be:

$$\tau_a \longrightarrow \tau_b \longrightarrow \tau_a \longrightarrow \tau_b \longrightarrow \ldots$$

The net result is as follows: within a 2T time interval, the first iteration of τ_b can miss the original deadline. The runtime behavior is portrayed in Figure 10. The original task fails to tolerate single-frame overloads; i.e., its IO operations occur after the deadline (time T). However, the transformed task only executes internal operations after T, which preserves the event-based semantics.

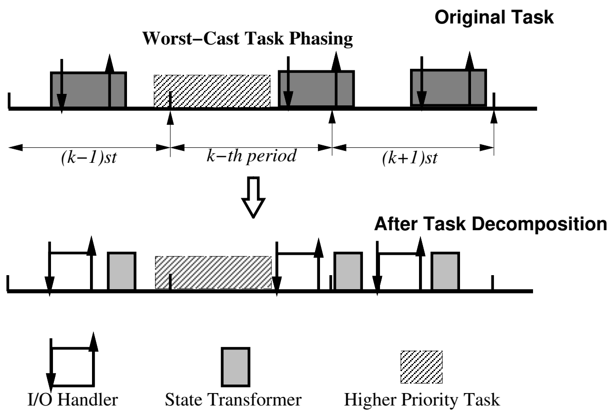

Figure 10: Scheduling of Newly Constructed Tasks.

Automatic Task Decomposition by Program Slicing. Straightforward as it may look, this decomposition can in reality be a very complex compiler problem. Many factors make this the case, among which are intertwined threads of control, nested control structures, complex data dependences between statements, procedure calls in the task code, etc. To cope with these problems in a systematic manner, we harness a novel application of *program slicing* [24, 29, 30].

Briefly stated, a *slice* of program P with respect to program point p and variable v consists of P's statements and control predicates that may affect the value of v at point p. To help carry out our slicing approach we use a *program dependence graph* [6, 12, 24], whose vertices represent a program's instructions, and whose edges represents control, data, and loop-carried dependences between them. For example, the program dependence graph corresponding to our controller program τ_3 is shown in Figure 11. Note that loop-carried dependences caused by "`state`" exist in the program dependence graph, even if there is no inner loop in the program: they derive from a loop-like nature of periodic tasks. We use the notation "$p \Rightarrow_* q$" to mean that node p can reach node q via zero or more control dependence edges or data dependence edges.

The slice of program P with respect to program point p and variable v – denoted as $P/\langle p, v\rangle$ – can be obtained through a traversal of P's program dependence graph. A simple algorithm to compute the slice is given below.

Algorithm 4.1 *Computes the slice $P/\langle p, v\rangle$:*

Step 1 Compute reaching definitions $RD(p, v)$ such that for any vertex $n \in RD(p, v)$, the statement n defines variable v, and control can reach p from n via an execution path along which there is no redefinition of v.

Step 2 Compute the slice by a backward traversal of the program dependence

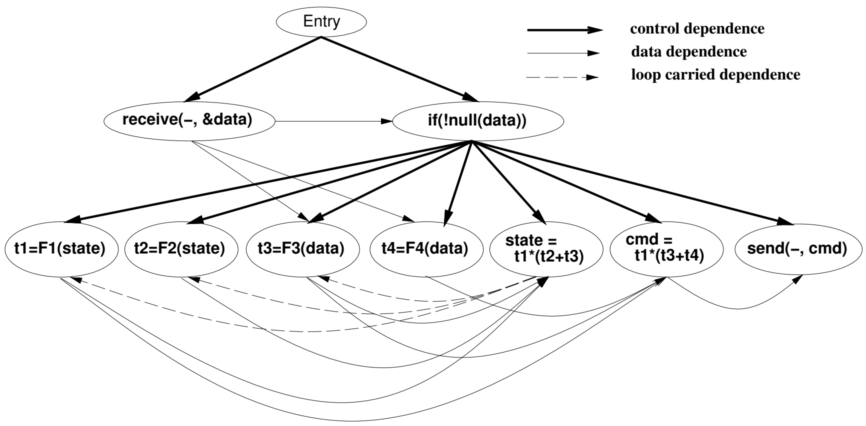

Figure 11: Program Dependence Graph for τ_3

graph such that

$$P/\langle p, v\rangle = \{m \mid \exists n \in RD(p, v) \ : \ m \Rightarrow_* n\}. \qquad \square$$

The definition of the program slice can be extended for a set of slicing criteria C in such a way that $P/C = \bigcup_{\langle p,v\rangle \in C} P/\langle p, v\rangle$.

Using the algorithm above, we can compute the subtasks τ_a and τ_b as follows:

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

Step 1 Compute the slice τ/IO, where

$$IO = \{\langle o, var(o)\rangle \mid o \text{ is observable and } var(o) \text{ is a variable used in } o\}$$

Step 2 Compute the slice τ/ST, where

$$ST = \{\langle eot, s\rangle \mid s \text{ is a state variable}\}$$

where eot is considered a point at the end of the task, and where a state variable is any variable which causes a periodic loop-carried dependence.

Step 3 Delete from τ/ST all non-conditional statements common to both of the slices. The remaining code becomes τ_b. $\square$

Using our original program dependence graph in Figure 11, the result of **Step 1** is depicted in Figure 12 (Top), while the result of **Step 2** is depicted in Figure 12

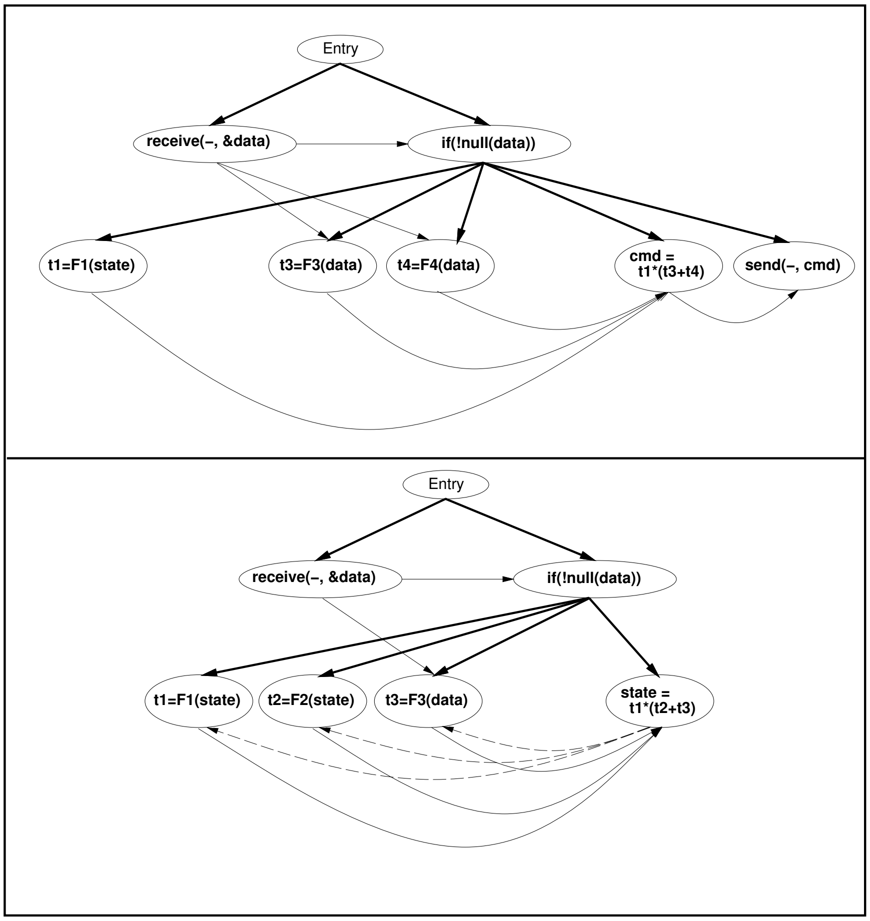

Figure 12: IO-Handling Slice (Top) and State-Update Slice (Bottom).

(Bottom). Returning to our task τ_3 in Figure 9, we see that L1, L3, L5, L6 and L8 are included in the IO-slice. Also, the predicate on line L2 is included, since the execution of L3, L5, L6 and L8 depend on its outcome. Similarly, L1, L2, L3, L4, L5 and L9 are included in the State-update slice. Then, by **Step 3**, common statements L1, L3 and L5 are deleted from it (but not L2!).

The textual results of these operations can be seen in Figure 13, where the total times for each subtask are:

$$C_{3a} = 4.93\text{ms}, \quad C_{3b} = 1.52\text{ms}$$

/* Subtask τ_{3a} */	/* Subtask τ_{3b} */

{
 receive(Sensor, &data); [0.50ms]
 c = !null(data); [0.06ms]
 if (c) [0.02ms]
   {
   t1 = F1(state); [1.05ms]
   t3 = F3(data); [1.35ms]
   t4 = F4(data); [1.35ms]
   cmd = t1 * (t3 + t4); [0.10ms]
   send(Actuator, cmd); [0.50ms]
   }
 }
	{
	 if (c) [0.02ms]
	   {
	   t2 = F2(state); [1.35ms]
	   state = t1 * (t2 + t3); [0.15ms]
	   }
	 }

Figure 13: Two Decomposed Subtasks.

4.3 Scheduling and Analysis

We can now sketch a high-level procedure that uses Algorithm 4.2 to transform an unschedulable task set into a schedulable one. The input is a set of n tasks

$$\Gamma = \{\tau_1, \tau_2, \ldots, \tau_n\}$$

which are processed in decreasing order, from τ_n to τ_1. If the task set is found to be unschedulable, Algorithm 4.2 is invoked to slice τ_n into its two constituent threads, which then replace τ_n in Γ. If the updated set is still deemed unschedulable, the procedure goes to work on τ_{n-1}, and so on.

In [9] we present a detailed alternative to this approach, in which tasks are processed from τ_1 to τ_n; i.e., the first task found unschedulable is selected for slicing. One can imagine other alternatives as well.

However, any such scheme is critically dependent on two elements:

(1) A scheduling policy that can exploit our task model; i.e. while the τ_b threads can miss their original deadlines, the precedence constraints between instances τ_a and τ_b must be maintained.

(2) An offline schedulability analyzer for the given scheduling policy.

There are several methods that can be used to address (1) and (2), each of which has its relative strengths and weaknesses. In this section we outline three such approaches which are useful for many applications, and also fairly simple to implement.

Method 1 (Slicing the lowest-priority task): We include this method to show that in many cases the scheduler need not be altered at all. Indeed, when the only lowest priority task is sliced – as in our example – the original rate-monotonic priority assignment may still be used. We illustrate this by returning to the example.

Online Scheduler: Now that τ_3's two subthreads have been isolated, they are "glued" back into a single task τ_3' by way of sequential composition:

$$\tau_3' = \tau_{3a}; \tau_{3b}$$

The new task τ_3' is still initiated every T_3 time units, and remains at the lowest priority. However, if iteration k has not completed when the new period $k+1$ starts, the new instance waits for the old one to finish.

Offline Analyzer: There is a simple method to determine whether the transformed task set is schedulable under this policy. First, since the higher-priority tasks remain unaltered, their schedulability can be determined using the "critical-instant" approach outlined in Example 4.1. As for the entire application, consider the newly constructed tasks in our example:

Task	Execution Time	Period
τ_1	$C_1 = 4.00$	$T_1 = 10$
τ_2	$C_2 = 4.00$	$T_2 = 16$
τ_3'	4.93 for τ_{3a} $+$ 1.52 for τ_{3b} $C_3' = 6.45$	$T_3 = 25$

In establishing schedulability, it is sufficient to check whether the following two conditions hold:

(1) whether τ_{3a} can always run within time T_3, and

(2) whether two successive instances of τ_3' can run within time $2T_3$.

An easy argument shows that this test is sound. Recall that task τ_3' is invoked at times 0, T_3, $2T_3$, etc. Let period i be the first in which τ_{3b} slides past its deadline. By condition (1), the i^{th} instance of τ_{3a} finishes by time $(i+1)T_3$, and by condition (2) the $i+1^{\text{st}}$ instance of τ_3' finishes by time $(i+2)T_3$. Thus task τ_{3a} finishes by time $(i+2)T_3$ as well, which preserves the TCEL semantics.

Returning to our example, we can verify conditions (1) and (2) by again making use of the critical-instant analysis. For condition (1), we "pretend" that τ_{3a} is autonomously invoked with a period of T_3, and show that it is schedulable with respect to tasks τ_1 and τ_2. Then, for (2), we assume another "imaginary" task $\tau_3'; \tau_3'$ – with period $2T_3$ and cost $2C_3'$ – and show that it too is schedulable with τ_1 and τ_2.

For τ_{3a}:	$3C_1 + 2C_2 + C_{3a}$	$=$	20.93	$<$	25	$=$	T_3
For $\tau_3'; \tau_3'$:	$5C_1 + 3C_2 + 2C_3'$	$=$	44.90	$<$	48	$=$	$3T_2$

Thus both conditions (1) and (2) hold. Note that this procedure yields a *sufficiency* test for schedulability, and not a necessary one. For example, the scheduler could, in fact, force *every* instance of τ_{3b} to miss its deadline. As long as τ_{3a} is always completed within the deadline, the TCEL semantics is maintained.

Method 2 (Slicing any number of tasks): Method 1 is clearly insufficient to dispatch several sliced tasks, each of which may have a different period. However, it can be generalized by slightly modifying the dispatching scheme. Briefly stated, when a task τ is sliced, it now receives *two* priorities: p_h, which is inversely proportional to its original period T, and p_l, which is inversely proportional to a period of $2T$. We briefly sketch the method below (details can be found in [9]).

Online Scheduler: Again, when a task τ is sliced, the two subthreads τ_a and τ_b are "glued" back into a single task "$\tau' = \tau_a;\ \tau_b$," where we let C_a and C_b denote the worst-case execution times of τ_a and τ_b, respectively. The scheduler uses a countdown timer for τ' to ensure that within any time-frame $[i \cdot T, (i+1) \cdot T]$, the first C_a units of execution time of τ' are run at priority p_h, with any remaining time in the frame run at priority p_l. Note that since τ_b may have "slid" beyond time $i \cdot T$, it may be the case that part of τ_b runs at priority p_h, while part of τ_a runs at priority p_l.

The algorithm works as follows: task τ' is initiated time 0 with priority p_h, and its countdown time is set to C_a. The timer decrements whenever τ' runs at priority p_h; when τ' gets preempted it is temporarily stopped. When the timer expires, τ' gets demoted to priority p_l. Note that this will always occur before time T if the IO-handler is to complete by its deadline.

As in Method 1, at time T task τ' is re-invoked, and if the old iteration is not finished the new one must wait. In either case, task τ''s priority is immediately reset to p_h, and the timer is reset to C_a as well. When the timer expires during the frame [T,2T], the priority is again demoted to p_l – *regardless of which iteration is currently running.* Thus, as we stated above, the net result is that during any given frame the task is first run at priority p_h for C_a time, with the remainder of time in the frame run at priority p_l.

The implementation of this scheme requires one countdown timer for each sliced task, which can easily be managed by only one programmable hardware timer – a standard component in most systems.

Offline Analyzer: We can subject task τ' to a simple schedulability test not unlike the one mentioned above. In doing so, we "pretend" that both tasks τ_a *and* τ_b are invoked autonomously every T time units, without regard to any precedence constraints. (Thus we pretend that that τ_a can always preempt τ_b.) The test involves checking the following two conditions.

(1) whether τ_a can always run within time T at priority p_h, and

(2) whether two successive instances of τ_b can run within time $2T$ at priority p_l.

This test – which actually assumes a static-priority, fully preemptive dispatching scheme – is still sufficient for our purposes. After all, condition (1) shows that when any task with cost C_a and priority p_h is released at times 0 and T, it will finish by time T and $2T$, respectively. Condition (2) shows that if another task with cost C_b and priority p_l is released at times 0 and T, both instances can finish by time $2T$. Since our online scheduler always raises τ''s priority at frame boundaries, and always lowers it after C_a execution time, the test's underlying notion of preemption simulates the behavior of the online scheduler.

Method 3 (A static-priority approach): Method 2 enjoys a simple analysis test for its dual-priority scheme, which is certainly one of its strengths. Its principal weakness is that the online component lacks the simplicity found in pure, static-priority scheduling. For example, recall that Method 1 successfully uses the original rate-monotonic scheduler.

Thus the following question arises: when can a set of transformed TCEL tasks be scheduled under a fully preemptive, static-priority scheme? Burns [2] provides an answer to this question after identifying a simple, but essential fact about the TCEL task model. That is, whenever we let a task's deadline be greater than its period, this represents a relaxation of the classical rate-monotonic restrictions put forth in [21]. Thus the rate-monotonic priority assignment may not be the optimal one; indeed, perhaps another static priority assignment will result in a feasible schedule when the rate-monotonic ordering fails.

Given a set of transformed TCEL tasks

$$\begin{aligned} \tau_1' &= \tau_{1a}; \tau_{1b} \\ \tau_2' &= \tau_{2a}; \tau_{2b} \\ &\vdots \\ \tau_n' &= \tau_{na}; \tau_{nb} \end{aligned}$$

it turns out the appropriate priority assignment is not only dependent on the periods (as in the pure rate-monotonic model), but also on the respective execution times of each IO-handler and state-update component. In [2] Burns presents a search algorithm to generate the feasible static-priority order – or to detect when no such order exists. Thus the approach includes the following components.

Online Scheduler: This is a simple, preemptive dispatching mechanism, in which priority "ties" are broken in favor of the task dispatched first. Thus, for example, a task's current iteration will finish before the next one starts.

Offline Analyzer: The analyzer is *constructive*, in that it produces a feasible priority assignment if one exists. If no such assignment exists, perhaps Method 2 can be used instead. In fact, we have identified task sets in which Method 2 succeeds and Method 3 fails, as well as other task sets in which Method 3 succeeds and Method 2 fails.

5 Conclusion

Modern real-time applications are becoming more complex in both their functional and temporal requirements, as well as their scale. At the same time, there is an increasing desire to use off-the-shelf hardware and standardized runtime support, with a concomitant demand for portability and upward compatibility.

These themes should not be surprising, since they also characterize the evolution of computer systems in general. In time-independent domains, however, they are often realized through the use of a high-level language, which serves as an abstraction between a designer's intentions and the low-level operation of the system architecture. Compilers, therefore, are relied on to do much of the dirty work, and to provide a bridge between the program and the platform.

The TCEL paradigm helps incorporate this development into real-time domains. As we have shown, TCEL's event-based semantics constrains only those operations that are critical to real-time operation; i.e., the events denoted in the specification or those derived from it. As such, a source program is an appropriate abstraction of the designer's intentions, and it need not over-burden the system with unnecessary constraints. Moreover, the event-based semantics enables a compiler to transform the program to suit the characteristics of the underlying system.

In this article we have presented two such transformation techniques. The first, trace-scheduling, helps resolve conflicts that arise when timing constraints are nested; e.g., when deadlines are tighter than periods, or when delay statements are inserted within the code. Our fine-grained synthesis procedure can be a useful tool for eliminating these conflicts. Since this is exactly the type of low-level work that compilers do best, a human programmer's time is probably better spent elsewhere.

The second technique is based on program-slicing, and it helps enhance the schedulability of periodic task sets. In this article we have concentrated on rate-based scheduling, one of the best understood areas in the real-time literature. However, the tradition has been to consider the "task" as an uninterpreted block of execution time – perhaps with a period, a start time and a deadline, but no other semantics to speak of. We have shown that once we "open up" the task to consider its event-based semantics, we can automatically convert an unschedulable application into a schedulable one. We believe that our approach can be used as a first-line defense in the tuning process, and is certainly preferable to measures such as hand-optimization or re-implementation in silicon – two of the more common remedies.

Acknowledgements

This research is supported in part by ONR grant N00014-94-10228, NSF grant CCR-9209333, an NSF Young Investigator Award CCR-9357850 and ONR/ARPA contract N00014-91-C-0195.

References

[1] A. Aiken and A. Nicolau. A development environment for horizontal microcode. *IEEE Transactions on Software Engineering*, pages 584–594, May 1988.

[2] Alan Burns. Fixed priority scheduling with deadlines prior to completion. Technical Report YCS 212 (1993), Department of Computer Science, University of York, England, October 1993.

[3] B. Dasarathy. Timing constraints of real-time systems: Constructs for expressing them, method for validating them. *IEEE Transactions on Software Engineering*, 11(1):80–86, January 1985.

[4] K. Ebcioglu and A. Nicolau. A global resource-constrained parallelization technique. In *International Conference on Supercomputing*, pages 154–163. ACM Press, June 1989.

[5] A. Mok et al. Evaluating tight execution time bounds of programs by annotations. In *Proceedings IEEE Workshop on Real-Time Operating Systems and Software*, pages 74–80, May 1989.

[6] J. Ferrante and K. Ottenstein. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9:319–345, July 1987.

[7] J. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computer*, 30:478–490, July 1981.

[8] F. Gasperoni. Compilation techniques for VLIW architectures. Technical Report RC 14915(#66741), IBM T. J. Watson Research Center, September 1989.

[9] R. Gerber and S. Hong. Semantics-based compiler transformations for enhanced schedulability. In *Proceedings IEEE Real-Time Systems Symposium*, pages 232–242. IEEE Computer Society Press, December 1993.

[10] M. G. Harmon, T. P. Baker, and D. B. Whalley. A retargetable technique for predicting execution time. In *Proceedings IEEE Real-Time Systems Symposium*, pages 68–77. IEEE Computer Society Press, December 1992.

[11] S. Hong and R. Gerber. Compiling real-time programs into schedulable code. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation.* ACM Press, June 1993. *SIGPLAN Notices, 28(6):166-176.*

[12] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graph. *ACM Transactions on Programming Languages and Systems*, 12:26–60, January 1990.

[13] Y. Ishikawa, H. Tokuda, and C. Mercer. Object-oriented real-time language design: Constructs for timing constraints. In *Proceedings of OOPSLA-90*, pages 289–298, October 1990.

[14] F. Jahanian and Al Mok. Safety analysis of timing properties in real-time systems. *IEEE Transactions on Software Engineering*, 12(9):890–904, September 1986.

[15] K. Kenny and K. J. Lin. Building flexible real-time systems using the Flex language. *IEEE Computer*, pages 70–78, May 1991.

[16] E. Kligerman and A. Stoyenko. Real-time Euclid: A language for reliable real-time systems. *IEEE Transactions on Software Engineering*, 12:941–949, September 1986.

[17] J. Krause. GN&C domain modeling: Functionality requirements for fixed rate algorithms. Technical Report (DRAFT) version 0.2, Honeywell Systems and Research Center, December 1991.

[18] I. Lee and V. Gehlot. Language constructs for real-time programming. In *Proceedings IEEE Real-Time Systems Symposium*, pages 57–66. IEEE Computer Society Press, 1985.

[19] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *Proceedings IEEE Real-Time Systems Symposium*, pages 166–171. IEEE Computer Society Press, December 1989.

[20] K. J. Lin and S. Natarajan. Expressing and maintaining timing constraints in FLEX. In *Proceedings IEEE Real-Time Systems Symposium*. IEEE Computer Society Press, December 1988.

[21] C. Liu and J. Layland. Scheduling algorithm for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, January 1973.

[22] A. Nicolau. *Parallelism, Memory Anti-aliasing and Correctness Issues for a Trace Scheduling Compiler*. PhD thesis, Yale University, June 1984.

[23] V. Nirkhe. *Application of Partial Evaluation to Hard Real-Time Programming*. PhD thesis, Department of Computer Science, University of Maryland at College Park, May 1992.

[24] K. Ottenstein and L. Ottenstein. The program dependence graph in a software development environment. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 177–184, May 1984.

[25] C. Park and A. Shaw. Experimenting with a program timing tool based on source-level timing schema. In *Proceedings IEEE Real-Time Systems Symposium*, pages 72–81. IEEE Computer Society Press, December 1990.

[26] L. Sha, J. Lehoczky, and R. Rajkumar. Solutions for some practical problems in prioritized preemptive scheduling. In *Proceedings IEEE Real-Time Systems Symposium*, pages 181–191. IEEE Computer Society Press, December 1986.

[27] A. Shaw. Reasoning about time in higher level language software. *IEEE Transactions on Software Engineering*, pages 875–889, July 1989.

[28] M. Smith, M. Horowitz, and M. Lam. Efficient superscalar performance through boosting. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 248–259. ACM Press, October 1992.

[29] G. Venkatesh. The semantic approach to program slicing. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, June 1991.

[30] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10:352–357, July 1984.

[31] V. Wolfe, S. Davidson, and I. Lee. RTC: Language support for real-time concurrency. In *Proceedings IEEE Real-Time Systems Symposium*, pages 43–52. IEEE Computer Society Press, December 1991.

[32] N. Zhang, A. Burns, and M. Nicholson. Pipelined processors and worst case execution times. *The Journal of Real-Time Systems*, 5(4), October 1993.