Compiling Real-Time Programs with Timing Constraint Refinement and Structural Code Motion *

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-3323, UMIACS-TR-94-90

July 1994

Abstract

We present a programming language called TCEL (Time-Constrained Event Language), whose semantics is based on time-constrained relationships between observable events. Such a semantics infers only those timing constraints necessary to achieve real-time correctness, without over-constraining the system. Moreover, an optimizing compiler can exploit this looser semantics to help tune the code, so that its worst-case execution time is consistent with its real-time requirements.

In this paper we describe such a transformation system, which works in two phases. First the TCEL source code is translated into an intermediate representation. Then an instruction-scheduling algorithm rearranges selected unobservable operations, and synthesizes tasks guaranteed to respect the original event-based constraints.

Keywords: Real-time, programming languages, compiler optimization, code scheduling, single-static assignment, timing analysis, trace scheduling, code motion.

*This research is supported in part by ONR grant N00014-94-10228, NSF grant CCR-9209333, and an NSF Young Investigator Award CCR-9357850. A preliminary abstract of this material appeared in the *Proceedings of the ACM SIGPLAN 93 Conference on Programming Language Design and Implementation* (June 1993).

1 Introduction

Developing a real-time system involves balancing a delicate equation. On one side are the functional requirements, which define valid translations from inputs to outputs. As such they are realized by programs, or *consumers* of time. On the other side are the temporal requirements, which place upper and lower time bounds between the inputs and outputs. Thus the requirements *constrain* time. When this equation fails to get balanced, the result is often a costly and arduous process of system tuning. This typically involves multiple, painful phases of instrumentation and hand-optimization. Additional measures may include re-coding key subsystems in assembly language, off-loading functions in programmable logic, or perhaps redesigning the system altogether.

Several programming languages help manage the requirements side of the design equation; examples are [13, 17, 19, 23, 27]. These languages provide programmers with a convenient means of postulating timing constraints within a program's text. The constraints are, in turn, conveyed to the real-time scheduler as a directive, or perhaps replaced by kernel calls to be invoked at runtime.

In this paper we present an automated methodology to help balance the other side of the design equation. This is done via two interrelated factors: a programming language and a compiler transformation engine. The real-time language is called TCEL (Time-Constrained Event Language), which contains timing constructs not unlike those in the abovementioned languages. However it differs significantly, in that its semantics is based on the time-constrained relationships between observable events. Since this imposes a "loose" interpretation of the constraints, a compiler can help rearrange the unobservable code to aid in the tuning process.

The distinction between events and local operations is not found in most real-time programming methodologies. Indeed, the standard approach is to establish timing constraints on *blocks of code*, without discriminating between the instructions within the code itself.

We have found this distinction to be quite useful. Since any relevant instruction can be annotated as event, the approach can be extended to most notions of "communication." For example, an event can be a message-passing operation, an access to memory-mapped I/O, an instruction that induces side-effects on other tasks, or for that matter, a reference to any designated function call or variable. (For simplicity, in the sequel we assume that only "**send**" and "**receive**" operations are observable.)

Consider the TCEL program fragment below, which receives sensor data, delays, receives more data; then it transforms the data into a command and sends out the result. The final **send** must take place within 4.0 ms of receiving the original message.

```
L1:  do
L2:    receive(p,&obj_coords1);
L3:  start after 3.5 ms finish within 4.0 ms
L4:    {
L5:    receive(p,&obj_coords2);
L6:    r1 = F(obj_coords1);
L7:    r2 = G(obj_coords2);
L8:    next_cmd = H(r1,r2);
L9:    send(q,next_cmd);
       }
```

In the usual interpretation of this program, statements L5-L9 would always execute after the delay; i.e., the **after** construct would be "executed" like a "sleep" command in Unix. TCEL's semantics, on the other hand, only induce timing constraints between the three event-triggering instructions: L2, L5 and L9. As for the unobservable statements L6-L8, their execution is bound only by natural control and data dependences.

This looser semantics yields an immediate benefit: if the execution times of L6-L8 conflict with the 4ms deadline, the unobservable code can be moved to help tune the program to its hardware environment. Indeed, perhaps all (or part) of L6 can be executed while the program delays. It may be possible to specialize parts of L7 and L8 to do the same; perhaps some pre-computations can even be executed before L1. In performing these transformations, the observable events act as "semantic markers," denoting the places where code can be moved.

In this paper we provide a means to carry out such transformations in a semi-automated fashion. Specifically, we are concerned with correcting *feasibility* faults, in which execution requirements conflict with the real-time constraints. While this objective is straightforward, achieving it is quite a difficult problem. Indeed, we show that even in the case of a basic block, determining the transformation strategy is NP-hard. And the situation gets much more complicated when the program possesses a branching structure, i.e., when actual execution paths are determined at runtime.

Thus we take a greedy approximation approach, which works in several phases. First the TCEL source is translated into a single-static-assignment (SSA) representation [3], whose naming conventions help isolate the "worst-case" execution paths. Next the code is decomposed into several blocks, and equations are generated to constrain their start and finish-times. Finally, a variant of trace-scheduling [6] is used to relocate the unobservable code, and hopefully attain feasibility.

The remainder of this paper is organized as follows. In the following section we survey related work in programming languages, semantics and optimization methods for real-time systems. Then, in Section 3 we present the syntax and semantics of TCEL, and follow it by Section 4, in which we introduce some preliminary notations. In Section 5 we give an overview of the scheduling problem, and briefly sketch our solution strategy. In Section 6 we discuss the decomposition phase, and the way we derive our code-based timing constraints. In Section 7 we present the code scheduling algorithm, which synthesizes feasible code from infeasible sections. We conclude in Section 8 with remarks on the enabling technologies we utilize, and the way we approach some of their limitations.

2 Related Work

TCEL's semantics was inspired by a principle commonly applied in formal methods. That is, when reasoning about a real-time concurrent system it is often useful to consider only "events of interest," and to abstract away local-state information. Indeed, almost all formal models ease this process by making some distinction between an "event" and a corresponding "action." For example, in Real-Time Logic [14], events are instantaneous – and require no resources – while actions consume nonzero time. Similar distinctions exist in RTRL [4], Timed IO Automata [21], ACSR [16], and in almost every formal approach to real-time.

It therefore seemed natural to extend this common technique to a "full-blown" real-time programming language, in which the "events" correspond to actual IO operations within C code. A logical consequence was the ability to exploit this looser semantics, and to use compiler transformations to move unobservable instructions out of over-constrained code blocks.

Most other real-time languages do not make such a distinction, and instead place constraints on the boundaries of code blocks. Two paradigms are used in these languages: either constraints are expressed directly in the program itself (as in [19, 17, 27]), or they are postulated in a separate interface, and then passed to the scheduler as directives. A common language-based approach (first presented in [17]) is to provide constructs such as "**within** t **do** $\{\ldots\}$," "**at** t **do** $\{\ldots\}$" and "**after** t **do** $\{\ldots\}$." An alternative, taken in [19], is to set up linear constraint expressions on the the start times and deadlines of code blocks. We have borrowed from both approaches: in the TCEL source we use the higher-level constructs, while in our intermediate code we make use of the constraint representation. But in TCEL the semantics is quite different, as it establishes constraints between the *observable events* within the code, and not on the code's textual boundaries.

There have been other compiler-based approaches to real-time programming [9, 10, 12, 23, 20]. These approaches, while addressing different problems associated with real-time programming, share a common goal, namely enhancing the predictability and schedulability of programs. In [10] a compiler classifies application program on the basis of its predictability and monotonicity, and creates partitions which have a higher degree of adaptability. The objective is to produce a transformed program possessing a smaller variance in its execution time. In [23] a partial evaluator is applied to a source program, which produces residual code that is both more optimized and more deterministic. In [28] an approach to speculative execution is postulated for distributed real-time systems. This is tangentially similar to our application of "speculative transformations," since both break control-dependences that are predicated on inputs. The principal objective in [28], however, is quite different, in that "shadow threads" are forked off to execute on available resources.

In [20] time-critical statements (or events) are assumed in the underlying programming language, and used for developing the notion of safe real-time code transformations. Based on this notion of safety, a large number of conventional code transformations are examined, and then classified for application in real-time programming. This approach comes closest to our work, in that the timing behavior of real-time programs is described in terms of events or the executions of time-critical statements. Unlike the semantics for TCEL, however, the execution times of non-time-critical statements are not explicitly decoupled from timing constraints imposed on the events. Thus, the applicability of some transformations may be unnecessarily restricted.

Finally, in [9] we describe how we use TCEL's event-based semantics to specialize tasks for multi-threaded applications. Specifically, a program-slicing tool splits a task into a deadline-sensitive sub-thread, and a thread which can be postponed past its deadline. Then a priority-based scheduling algorithm is used to postpone the execution of the second thread, thereby enhancing schedulability of the entire application.

3 The Language of Time-Constrained Events

In this section we present TCEL's constructs to express timing constraints within a program. Both constructs are syntactic descendents of the *temporal scope*, first introduced in [17]. However, as we have stated, our semantics is quite different, in that it relies on constrained relationships between observable events.

We first elaborate the "**do**" construct which establishes several types of relative timing constraints. Its general form is as follows.

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

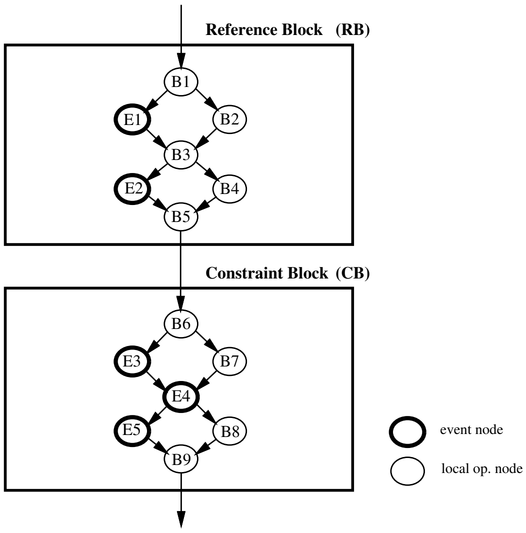

Figure 1: Typical Flow Graph.

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 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 executed 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. However, unlike the untimed counterpart, event operations cannot be part of the condition. 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. The periodic constraints established by this statement are depicted in Figure 2, where the time-line shows the first two instances of the statement.

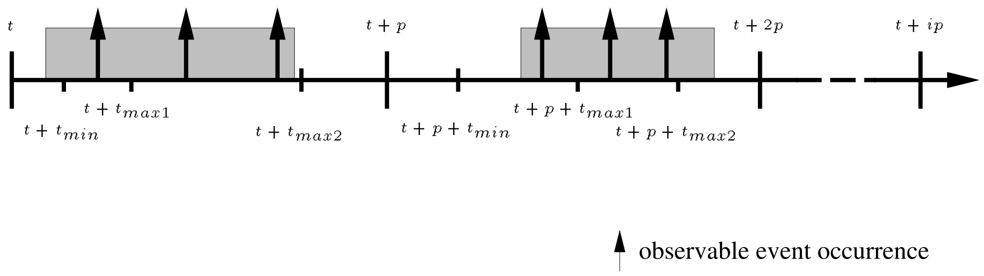

Figure 2: Behavior of Periodic Timing Construct.

Examining the time-line, we see that the **every** statement is released at time t, and that within the first frame, the first observable event (denoted by an arrow) occurs between $t + t_{min}$ and $t + t_{max1}$. Similarly, the first frame's last event occurs before $t + t_{max2}$. Generalizing, the following constraints are induced for period i:

- **start after** t_{min}: The first event executed in the CB occurs after $t + (i - 1)p + t_{min}$.
- **start before** t_{max1}: The first event executed in the CB occurs before $t + (i - 1)p + t_{max1}$.
- **finish within** t_{max2}: The last event executed in the CB occurs before $t + (i - 1)p + t_{max2}$.

As we have stated, timing constructs may be arbitrarily nested. Consider the program in Figure 3(A), which is a (very gross) 2-dimensional abstraction of an aircraft navigation/control loop. A set of route coordinates are maintained in the array "`GOAL`," which is maintained by another module. The TCEL program's role is to (1) sample the aircraft's current coordinates, its (true) heading, roll, and its ground speed; (2) get the next route coordinate to visit; (3) compute the relative attitude between the heading and the coordinate; and (4) adjust the course by updating throttle and roll. Adjustments are made in discrete increments, and are contingent on the *current* roll and velocity, as well as the amount that the course must be changed.

The timing constraints are induced as follows:

(1) Control updates are made periodically, with rate 50/second.

(2) In order to give the actuators time to get updated (and for the craft to respond accordingly), all updates must be made within the first 5 ms of each period.

(3) Velocity (ground speed) is obtained via a "request-response" protocol from an external unit; the response arrives with maximum latency of 0.75 ms.

(4) To correlate ground speed with outputs, all throttle and flap updates must be made within of 3.1 ms of *actual* ground speed sample. In the best case this may be made upon issuing the request.

If the specification mandated additional timing constraints, clearly we could employ further levels of nesting to achieve them. For example, suppose we desired to add a fifth timing requirement to the four listed above:

(5) The final two outputs must be correlated within 0.5 ms of each other. (This is not unrealistic, since the two outputs control are coordinated to effect the angular adjustment.)

To accomplish this we would replace the sequential composition with an additional nested TCEL statement:

do
    output(THROT, throttle);
finish within 0.5 ms
    output(FLAP_Cntrl, wflap);

The net runtime effect would simply be a refinement of the potential behaviors; i.e., the time-event relationships exhibited by the altered program would be a subset of those in the original version.

4 Basic Notations

The output of TCEL compiler's machine-independent pass is a flow graph, which contains the original timing information. For example, Figure 3(B) shows the flow graph for our flight control program in Figure 3(A), where for the sake of brevity we have left the code in its original C form.

We call this structure a *hierarchical flow graph*, since it maintains the program's original hierarchical levels of scoping.

Definition 4.1 A *hierarchical flow graph* $HFG(\mathrm{B})$ of code block B is a directed graph

$$(V, E, entry(\mathrm{B}), exit(\mathrm{B}))$$

where V is a set of nodes, E is a set of edges representing control flow, and where $entry(\mathrm{B}) \in V$, $exit(\mathrm{B}) \in V$ denote the unique entry and exit of B, respectively.

A vertex $n \in V$ may be either a basic block of B, an entry node, an exit node, or another hierarchical flow graph $HFG(\mathrm{B'})$. $\square$

Thus all nested constructs, including loops, are reduced into single nodes. Moreover, since we do not move code out of loops – but only within their bodies – we can treat a *HFG* as an acyclic graph, and ignore all back edges.

Of course our hierarchical structure assumes that the program's (flat) flow graph *reducible* [1]. But since "structured" programs without unrestricted **goto**s lead to reducible flow graphs, without loss of generality we assume that our programs possess this property.

```
every 20 ms finish within 5 ms
  do
  {
    /* Get current position, heading & roll */
    input(GPS, &x, &y);
    input(NAV, &theta);
    input(IMU, &roll);

    /* Request current velocity */
    output(Cntrl_out, REQ_VEL);
  }
  start after 0.75 ms finish within 3.1 ms
  {
    /* Get current velocity */
    input(Cntrl_in, &vel);

    /* Update target position */
    if (GOAL[i].passed) {
      gx = GOAL[i].x;
      gy = GOAL[i].y;
      i = (i+1) % NCOORD;
    }

    /* Using relative attitude w.r.t target, */
    /* compute angular adjustment.           */
    rtheta = compRelAtt(theta, x, y, gx, gy);
    if (|rtheta| < EPS)
      dtheta = 0.0;
    else {
      if (vel < VHIGH)
        dtheta = rtheta;
      else
        dtheta = safeDtheta(rtheta, roll);
    }

    /* Adjust flap and throttle for heading. */
    wflap = compFlapw(roll, vel, dtheta);
    throttle = compThrot(roll, vel, dtheta);
    output(THROT, throttle);
    output(FLAP_Cntrl, wflap);
  }
```

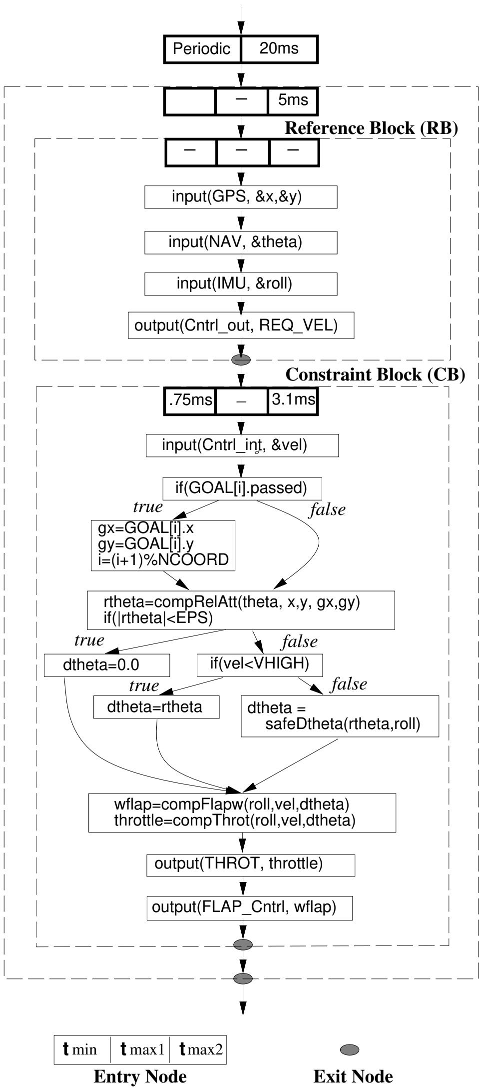

Figure 3: (A) Source Code for Flight Controller, and (B) Flow Graph.

For a given block B, let $HFG(\mathrm{B}) = (V, E, entry(\mathrm{B}), exit(\mathrm{B}))$. We can easily extend traditional compiler terminology to $HFG(\mathrm{B})$ as follows.

Paths. A path (or trace) between $n_1, n_2 \in V$ is denoted by "$n_1 \rightarrow^b n_2$," where b is the sequence of nodes traversed between (and including) n_1 and n_2. When b includes a node $m \in V$, we denote this by overloading the set membership operator, i.e., as "$m \in b$."

We also use the path relation, but omit the actual path, to denote the existence of *some* path between two nodes, i.e.,

$$n_1 \rightarrow n_2 \stackrel{\text{def}}{=} \exists\ b : n_1 \rightarrow^b n_2 \qquad \square$$

Data-dependence. For a node $n \in V$, we define $Def(n)$ and $Use(n)$ to be sets of variables defined by n and used in n, respectively. Thus, for nodes $n_1, n_2 \in V$, n_2 is *data dependent* on n_1 (denoted "$n_1 \stackrel{d}{\rightharpoonup} n_2$") iff there is a path b such that $n_1 \rightarrow^b n_2$ and

$$\exists v \in Def(n_1) \cap Use(n_2) :: (\forall n \in V :: (n \in b) \wedge v \in Def(n)) \Rightarrow n = n_1)$$

For nodes n_1 and $n_2 \in V$, we say that n_2 is *transitively* data dependent on n_1 (denoted "$n_1 \stackrel{d}{\rightharpoonup}_+ n_2$") iff there is a path

$$n_1 \stackrel{d}{\rightharpoonup} n_1' \stackrel{d}{\rightharpoonup} n_1'' \stackrel{d}{\rightharpoonup} \ldots \stackrel{d}{\rightharpoonup} n_2 \qquad \square$$

Control dependence. For nodes $n_1, n_2 \in V$, n_2 is *control dependent* on n_1 (denoted "$n_1 \stackrel{c}{\rightharpoonup} n_2$") iff n_1 represents a control predicate and n_2 is immediately nested within the loop or conditional whose predicate is represented by n_1.

Dependence Closure. The *dependence closure* for node n in the block B, denoted by "$DC(n, B)$," contains n and all nodes m that reach n via zero or more control or data dependence edges. It is inductively defined by the following least fix-point operation:

$$\begin{array}{rcl} DC(n, B) & = & fix\, F(\{n\}) \cup \{n\},\ \text{where} \\ F(S) & = & \{m \in V \mid \exists\, n' \in S : m \stackrel{d}{\rightharpoonup} n' \ \vee m \stackrel{c}{\rightharpoonup} n'\} \end{array}$$

When we are concerned only with data dependences, we make use of *data dependence closure*, "$DDC(n, B)$" defined as below.

$$DDC(n, B) = \{m \in V \mid m \stackrel{d}{\rightharpoonup}_+ n\} \ \cup \{n\} \qquad \square$$

5 The Problem and Our Solution

TCEL provides a powerful framework for describing real-time programs at the source-code level. The timing constructs allow a programmer to express the application's real-time constraints in a straightforward manner; moreover, the event semantics provides an unambiguous interpretation of the constructs. However, two fundamental issues remain, which are endemic to all real-time systems.

1. A program may not be *feasible*, i.e., a single process's execution time may conflict with its own real-time constraints.

2. While the program may be feasible, it may not be *schedulable* under any tractable real-time scheduling algorithm.

As for the feasibility problem, consider the following TCEL fragment:

```
do
    input(P, &m);
start after 10ms finish within 20ms
{
    input(Q, &x);
    S;                              [20ms]
    output(R, y);
}
```

The code's timing constraints mandate a 10ms latency between the events generated by "**input**(P, &m)" and "**input**(Q, &x)," as well as a 20ms deadline between the events generated by "**input**(P, &m)" and "**output**(R, y)." Meanwhile, the bracketed "20ms" denotes that the unobservable statement S requires a maximum of 20ms to execute, a bound obtained by a timing analysis tool (e.g., [11, 18, 24, 25, 29]). Consequently, the program possesses an inherent conflict, since S requires 20ms to execute while it is only allowed 10ms.

We address this problem by an approach we call *feasible code synthesis*. In our example this would involve decomposing S and, if possible, moving instructions not dependent on "x" out of the overloaded section. However merely achieving feasibility may be of little help, since the transformed code may still not be schedulable under any known methods. For example, when a program contains if-then-else branches, then the actual execution paths (and the events executed) are determined dynamically. But since schedulers must provide guarantees, they do not have the flexibility to instantaneously, dynamically reschedule a task set whenever an event is triggered. Indeed, while an event-based semantics makes conceptual sense at the source-program level, most real-time schedulers only accept timing constraints on the start and finish times of *tasks*.

Since the strategies used to achieve feasibility have a profound affect on the ultimate task structure, we solve these two problems together. Thus the role of the TCEL compiler is to partition event-driven source programs into time-constrained blocks of code, in which all of the blocks are feasible.

5.1 The Problem of Feasible Code Synthesis

Even without the task partitioning component, achieving feasibility is a nontrivial problem. This is obviously the case if we allowed potentially unbounded loops (or conditional goto's), which would render the problem undecidable. But since real-time programs must be amenable to worst-case timing estimates, we assume that upper bounds can be obtained for execution times. Formally, we let "$wt(S)$" denote a statement's worst-case execution time, where we assume that $wt(S) \in [0, \infty)$ for any statement S. Obviously wt is implementation dependent, and its tightness is determined by quality of the analysis tool used to generate the bound.[1]

While the feasibility problem may be decidable for our domain, it is not necessarily trivial. Even when program is structured like our example above, and where S is a *basic block*, simply deciding whether the program can be made feasible is still NP-hard.

The problem can be stated as follows: given a TCEL timing construct

$$\textbf{do}\ \mathrm{RB}\ \ \textbf{start after}\ t_{min}\ \ \textbf{start before}\ t_{max1}\ \ \textbf{finish within}\ t_{max2}\ \ \mathrm{CB}$$

is it possible to transform RB and CB to meet the following constraints?

(1) On any execution path, the original ordering of observable events is maintained.

(2) The original data and control dependences are preserved between instructions and events.

(3) The code's execution time does not conflict with the timing constraints between the events.

Clauses (1)-(2) imply that the transformed code must be functionally correct: events may not be reordered, and the original relationships between input and output data must be maintained. Clause (3) means that the new code is feasible.

This problem is NP-hard, due to the existence of immovable operations and data dependences.

Theorem 5.1 *Feasible code synthesis is NP-hard.*

Proof: The proof follows by a straightforward transformation from "Partition[SP12]" [7] to feasible code synthesis. Consider an instance (A, s) of Partition, where $A = \{a_1, a_2, \ldots, a_n\}$ is a set of elements, and where $s : A \mapsto \mathbb{N}$ is the cost of each element. Letting $\sum_{i=1}^{n} s(a_i) = 2T$, a partition of A is some $A' \subseteq A$ such that $\sum_{a \in A'} s(a) = T = \sum_{b \in A - A'} s(b)$. Determining whether such an A' exists is equivalent to determining whether the following TCEL program can be made feasible:

[1] We return to this issue in the Conclusion.

```
        do
E1:         input(P, &x);
        start after T finish within 2T
        {
E2:         output(Q, g(x));
L1:         x1=f1(x);                          [s(a1)]
L2:         x2=f2(x);                          [s(a2)]
              ⋮                                   ⋮
Ln:         xn=fn(x);                          [s(an)]
E3:         output(R, h(x1, ..., xn));
        }
```

Here $E_1 - E_3$ generate events, $L_1 - L_n$ are unobservable instructions, and each line is considered atomic (that is, a line must be relocated as a single entity). Then by the construction,

(1) E_2, $L_1 - L_n$ are mutually data independent.

(2) E_2, $L_1 - L_n$ are data dependent on E_1.

(3) E_3 is data dependent on $L_1 - L_n$.

If we assume that $wt(L_i) = s(a_i)$ for $1 \leq i \leq n$, and that $wt(E_j) = 0$ for $1 \leq j \leq 3$, then there exists a partition of our original set A if and only if there is a feasible transformation for the program.

As for the "if" part, assume there is a partition $A' \subseteq A$. Then for all $a_i \in A'$, moving the corresponding instruction L_i between the events E_1 and E_2 ensures feasibility: exactly T execution time is consumed between E_1 and E_2, and another T is used between E_2 and E_3.

As for the "only if" part, assume there is a feasible transformation. Then, since $2T$ execution time is used overall, the constraints mandate that at most T of it be used between E_1 and E_2, and the rest between E_2 and E_3. Thus we must move some set of instructions $L \subseteq \{L_1, \ldots L_n\}$ between E_1 and E_2, where $\sum_{L_j \in L} wt(L_j) = T$. But then the corresponding elements in A form a partition. □

When both the constraint block and reference block consist of straight-line code, the problem is obviously in NP as well. A feasible ordering can always be "guessed" and then verified, which consequently yields the following corollary.

Corollary: Feasible code synthesis is NP-complete for straight-line code. □

5.2 Solution Strategy

In proving Theorem 5.1 we used the simplest possible TCEL program, which possess just two basic blocks. In this case a feasible transformation would simply reorder the instructions, while keeping the program's fundamental structure intact.

But the situation gets significantly more complicated when the program possesses branches, and when the events that get executed are determined at runtime. Since attaining feasibility mandates that we statically guarantee the timing constraints along *all* execution paths, reordering the instructions along a *single path* may not be sufficient. In the worst of all cases, all paths of a multi-branching program would have to be "expanded out," and then each one reordered individually – potentially requiring time *and* space exponential in the original number of instructions. Clearly this is not an attractive solution approach. Moreover, it would render the problem of *schedulable* task decomposition – our second objective – next to impossible.

Thus we take the following alternative approach, in which feasible tasks are synthesized in a two-step process – *section decomposition* (Section 6) and *code scheduling* (Section 7).

Section Decomposition. First the code is translated into its single-static-assignment (SSA) form [3]. This representation serves two purposes: (1) it yields a compact means of representing the data-dependence relation discussed in Section 4; and (2) SSA's convention of uniquely naming each variable assignment is precisely what we require in the code transformations phase. (A review of SSA's base conventions are given in the Appendix.)

Next, the timing construct is decomposed into *sections*, which represent the natural schedulable "task" units. This step involves determining the section boundaries, as well as generating a set of dispatch equations that constrain the start-times and finish-times of each section.

Code Scheduling. In this step the dispatch equations are checked for their consistency with the code's execution time. If there is an inconsistency, program transformations are used to relocate the unobservable code across section boundaries.

The algorithm is a greedy approximation, in that each section is processed locally, with the goal of attaining consistency for an entire program. The following strategy is used: if a given section is determined to be overloaded, code is moved out of the section and "up" on the *HFG*. After the section at hand is determined consistent, the target section can be processed in turn. Thus the net result is an "upward migration" of unobservable instructions, which terminates either when the program achieves consistency, or it is determined to be inconsistent.

An analogous strategy is used for programs with nested timing constructs. In this case the *HFG* is scheduled in a "bottom-up" manner. That is, the innermost nodes in the *HFG* are scheduled first, with the goal of satisfying their local constraints. Then, the surrounding node is handled. If *this* level is found inconsistent, the inner nodes are "opened up" once again, and more aggressive optimization is carried out.

The actual transformations are similar to those used in Trace Scheduling [6]. As the name implies, the Trace Scheduling algorithm works on specific traces: it selects a path (or trace) from a given code block, and then selects instructions on that path to move. Such an approach is well suited to our purpose, because it can focus on the traces which have the maximum execution time among all traces within a given code block.

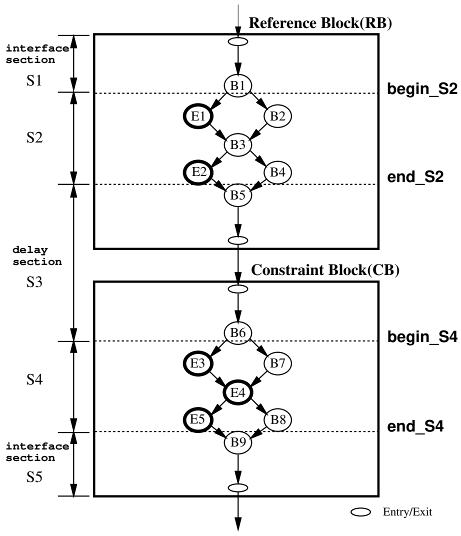

Figure 4: The Flow Graph of a Timing Construct and its Section Division.

6 Section Generation

Section generation is the process of decomposing the program into a set of "tasks" (or *sections*). The input is the original *HFG* (e.g, that portrayed in Figure 3(B)), with the output being a slightly different *HFG*, which is more amenable to real-time dispatching. This involves dividing a timing construct 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). 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.

6.1 Determining Section Boundaries

Recall the discussion of the $FIRST$ and $LAST$ functions in Section 3. 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 $LAST(\text{RB}) \cup \{exit(\text{RB})\}$.

- 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 $LAST$(CB) and the entry node. Thus, its logical place 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 flight controller program from Figure 3. Figure 5 illustrates its constituent sections. The constraint-expression for S6 corresponds to the program's outer, periodic loop. As the program is in SSA form, ϕ-functions appear at confluence points where different values of the same variable in the original program merge. For example, "`dtheta4=`ϕ`(dtheta1,dtheta2,dtheta3)`" is inserted at the place where three different values of `dtheta` merge. As a result, each use of a variable is reached by a unique assignment. Again, the bracketed numbers denote the maximum execution times of the corresponding operations on the targeted CPU. On modern architectures, fine-grained operations like simple assignments possess minuscule execution times, and cannot be measured (in isolation) by any timing tool. For the sake of presentation we assume that such instructions take zero time, and concentrate on larger-grained function calls and the like. [2]

6.2 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 3.1ms, while in Figure 5 it is tightened to 3.0ms. 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

[2] We revisit this issue in the Conclusion.

```
S6:  (S6.start[p]≥p×20ms, S6.finish[p]≤p×20ms+5ms)
     {
S1:    {
         gx1 = φ(gx3, gx0);
         gy1 = φ(gy3, gy0);
         i1 = φ(i3, i0);

         input(GPS, &x1, &y1);                                  [.1ms]
         input(NAV, &theta1);                                   [.1ms]
         input(IMU, &roll1);                                    [.1ms]
       }
S2:    {
         output(Cntrl_out, REQ_VEL);                            [.1ms]
       }
S3:    { /* Null */ }
S4:    (S4.start≥S2.finish+0.75ms, S4.finish≤S2.finish+3.0ms)
       {
         input(Cntrl_in, &vel1);                                [.1ms]
         c1 = GOAL[i1].passed
         if (c1) {
           gx2 = GOAL[i1].x;
           gy2 = GOAL[i1].y;
           i2 = (i1+1) % NCOORD;
         }
         gx3 = φ(gx2, gx1);
         gy3 = φ(gy2, gy1);
         i3 = φ(i2, i1);

         rtheta1 = compRelAtt(theta1, x1, y1, gx3, gy3);        [.25ms]
         c2 = |rtheta1| < EPS;
         if (c2)
           dtheta1 = 0.0;
         else{
           c3 = vel1 < VHIGH;
           if (c3)
             dtheta2 = rtheta1;
           else
             dtheta3 = safeDtheta(rtheta1, roll1);              [.43ms]
         }
         dtheta4 = φ(dtheta1, dtheta2, dtheta3);

         wflap = compFlapw(roll1, vel1, dtheta4);               [.95ms]
         throttle1 = compThrot(roll1, vel1, dtheta4);           [.89ms]
         output(THROT, throttle1);                              [.1ms]
         output(FLAP_Cntrl, wflap);                             [.1ms]
       }
S5:    { /∗ null ∗/ }
     }
```

Figure 5: Flight Control 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:

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

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:

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

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 3.1ms to 3.0ms.

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:

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

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

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

Table 1: Timing Constraints of S3 and S4.

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

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:

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

These are the necessary and sufficient conditions to achieve feasibility, and they are summarized in Table 1.

Returning to our example, we find that section S4 violates its duration constraint ($DUR(\mathrm{S4}) =$ 2.25ms),[3] since $wt(\mathrm{S4})$ by far exceeds it. (Adding up the time annotations yields $wt(\mathrm{S4}) = 2.82ms$ along the worst-case execution time path.) In the next subsection we discuss our code-scheduling techniques which handle cases such as this, in which the duration constraints fail to hold.

7 Code Scheduling

The code scheduling algorithm is inspired by a common compiler strategy used for VLIW and superscalar architectures [2, 5, 6, 8, 22, 26]. In such domains, an optimizing compiler exploits a program's inherent fine-grained parallelism, and "packs" its computations into as many functional units as possible. Thus the objective is to keep each unit busy, and to achieve better overall throughput.

Our problem context has an entirely different goal, and it cannot be solved by *directly* applying well-known techniques such as Trace Scheduling [6] or Percolation Scheduling [2, 5]. We are concerned not with enhancing average-case performance, but instead with ensuring feasibility. In fact, we will be satisfied with even *increasing* the program's overall execution time – as long as the timing constraints are met.

[3] $T_{max_2} - T_{min} = 3.0\mathrm{ms} - 0.75\mathrm{ms} = 2.25\mathrm{ms}$

```
algorithm Code_Scheduling(T) /* T is a timing construct */
input: the ordered set of sections {S1, S2, ..., S5} in T
begin
    d = T_max2 − T_min;
    /* Schedule code from S4 into S3 */
    call Schedule_Section(S4, S3, d, ∅);
    recompute T_max1; /* to reflect the change in Δ_S4 */
    if (wt(S3) ≤ T_min) then exit("No scheduling needed for S3.");
    else d = min{T_max1, T_max2 − wt(S4)};
    /* Schedule code from S3 into S1 */
    call Schedule_Section(S3, S1, d, Def(S2));
end
```

Figure 6: Top-level Algorithm for Code Synthesis.

7.1 The Top-Level Algorithm

Our approach to code scheduling is a greedy approximation, and it attempts to attain the desired consistency of a timing construct in a section-by-section manner. It inspects sections S4 and S3 (in reverse topological order), and checks whether they satisfy their duration constraints. If S4 violates its constraint, the algorithm attempts to reduce its surplus execution time by moving nodes into section S3. In turn it processes section S3, which may now contain newly moved code.

To perform greedy code motion, we have adapted a technique from the approach to Trace Scheduling in [6], and we use it as a component of the code scheduling algorithm. In our approach, nodes lying on paths that exceed their section's duration constraint are considered for code motion. We distinguish such paths as *critical traces*. Formally, a critical trace t of section S is defined as a path

$$entry(\mathrm{S}) \rightarrow^{t} exit(\mathrm{S})$$

such that $wt(t) > DUR(\mathrm{S})$. The reason we use the trace-based approach is straightforward: optimizing to avoid hard real-time exceptions demands scheduling only the critical traces, and no others.

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, as in our example).

The top-level algorithm calls subroutine "*Schedule_Section*," which then schedules the overloaded section at hand. Note that when code is scheduled from S3 into S1, the variables defined in S2 are passed to the subroutine, which ensures the dependences from S2 to S3 are maintained. In the following subsection we discuss in detail the innards of this subroutine, and the strategies it uses to solve the scheduling problem.

7.2 Subroutine `Schedule_Section`(S,D,DUR(S),V_{bar})

For the flow graphs of source section S and destination section D, the critical trace scheduling problem is to construct new flow graphs for S and D such that:

(1) The observable nodes of S remain in S and keep their relative order on any paths in S.

(2) $wt(t) \leq DUR(\mathrm{S})$ holds for all traces t in S.

(3) Execution ordering established by code's original data dependences is preserved.

When code is scheduled from S3 into S1 the parameter V_{bar} – containing the variables defined in S2 – is required to help maintain property (3).

As we have stated, the trace-based approach is attractive precisely because it allows us to concentrate on paths which violate the duration constraints. However, a direct application of Trace Scheduling induces a severe liability – extra code must be inserted to preserve the program's semantic integrity. In the parlance of instruction-scheduling, this is typically called *bookkeeping code*.

Consider the SSA program in Figure 7(B). Since the instruction "`z1=F(x)`" is free of a data dependence on the variable"`y`," it *may* be eligible to be moved into S3. (This transformation – which we develop in the sequel – is called *speculative*, since it breaks a control dependence.) As shown in Figure 7(C), moving the instruction requires no additional code to maintain the program's semantics; SSA's naming conventions maintain the correctness.

However a more aggressive policy could be carried out, which is shown in Figure 7(D). Note that additional code may be moved without breaking data dependences; even the variable `r` may be split into movable parts (i.e., `r1` and `r2`) and the parts that depend on `y` (i.e., `r3` and `r5`). However, the price we pay is the additional bookkeeping code required to maintain correctness.

One obvious problem with bookkeeping is that it induces a significant amount of extra code – indeed, if carried to extremes, the transformations in Figure 7(D) may result in an exponential blow-up. And in our problem context bookkeeping may have an additional, "fatal" effect:

Scheduling a critical trace may insert bookkeeping code on other, non-critical traces, and thereby increase their execution times. Hence a non-critical trace may become critical.

To avoid this drawback of Trace Scheduling, we use the type of transformation depicted in Figure 7(C), and we use it as aggressively as possible. The strategy involves repetitively applying

<table>
<tr><th>TCEL Source</th><th>SSA Form</th><th>Our Approach</th><th>Bookkeeping Approach</th></tr>
<tr>
<td>do
  input(P, &x);
start after t_1
 finish within t_2
{
  input(Q, &y);
  if p(y) {
    a = E(y);
    z = F(x);
  } else
    z = G(y);
  if q(y);
    r = H(z);
  else
    r = K(z);
  s = I(r);
   ⋮
}</td>
<td>⋮
S2: {
    input(P, &x);
  }
S3: { /* Null */ }
S4: (S4.start≥ . . . ,
    S4.finish≤ . . .)
  {
    input(Q, &y);
    if p(y) {
      a1 = E(y);
      z1 = F(x);
    } else
      z2 = G(y);
    a = ϕ(a1, a0);
    z = ϕ(z1, z2);
    if q(y)
      r1 = H(z);
    else
      r2 = K(z);
    r = ϕ(r1, r2);
    s = I(r);
     ⋮
  }</td>
<td>⋮
S2: {
    input(P, &x);
  }
S3: {
    z1=F(x);
  }
S4: (S4.start≥ . . . ,
    S4.finish≤ . . .)
  {
    input(Q, &y);
    if p(y)
      a1 = E(y);
    else
      z2 = G(y);
    a = ϕ(a1, a0);
    z = ϕ(z1, z2);
    if q(y)
      r1 = H(z);
    else
      r2 = K(z);
    r = ϕ(r1, r2);
    s = I(r);
     ⋮
  }</td>
<td>⋮
S2: {
    input(P, &x);
  }
S3: {
    z1 = F(x);
    r1 = H(z1);
    r2 = K(z1);
  }
S4: (S4.start≥ . . . ,
    S4.finish≤ . . .)
  {
    input(Q, &y);
    if p(y) {
      a1 = E(y);
      if q(y) { }
      else { }
      r3 = ϕ(r1, r2);
    } else {
      z2 = G(y);
      if q(y)
        r4=H(z2);
      else
        r5=K(z2);
      r6 = ϕ(r4, r5);
    }
    a = ϕ(a1, a0);
    z = ϕ(z1, z2);
    r = ϕ(r3, r6);
    s = I(r);
     ⋮
  }</td>
</tr>
</table>

Figure 7: (A) Source, (B) SSA Form, (C) Bookkeeping-free Transformations and (D) Bookkeeping

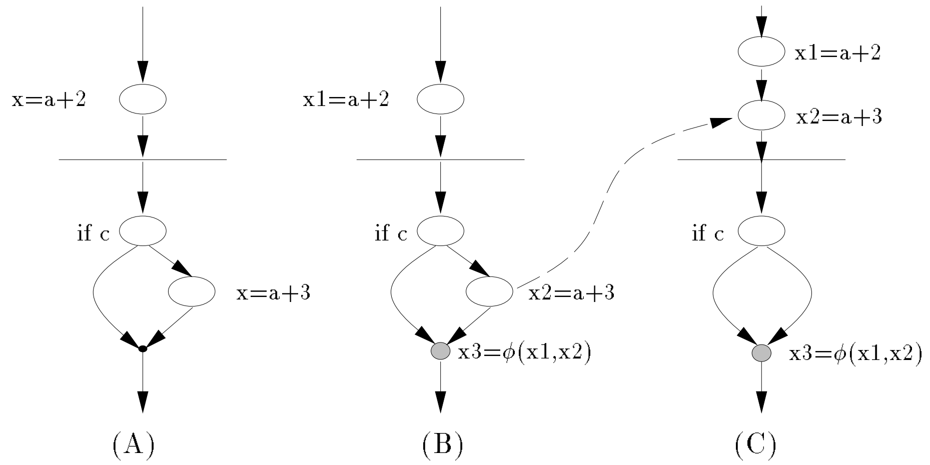

Figure 8: Speculative Code Motion: (A) Original Code, (B) SSA Code and (C) Transformed Code.

the following three steps: (i) finding a critical trace t; (ii) identifying a node n which can be moved into the destination section D; and (iii) moving n into D, along with n's ancestor nodes required to maintain the program's semantics. Since our objective is to keep the amount of new code to a minimum, step (iii) translates into the following rules for moving n into D:

(1) n's data dependence predecessors are moved along with n; *i.e.*, the nodes on which n is transitively data dependent.

(2) The control-dependence predecessors (i.e., the **if-then-else**'s guarding n) are treated as follows:

(a) If possible they are copied into D, so that they still guard the execution of n.

(b) Otherwise (as in Figure 7(C)), n will now be unguarded in its destination section D.

Thus the end result of code scheduling appears as if large-grained control structures were rearranged, and hence we name the strategy *structural code motion*. Yet code scheduling is still trace-based, since it is driven by worst-case paths.

Consider a node n to be moved into the destination section. When *all* of n's dependence predecessors (both data and control) are moved (or copied) along with n, the new execution ordering is guaranteed to maintain the program's original semantics.

But what are the ramifications of case 2(b) above, i.e., where control dependences are broken? Consider Figure 8(A), where we assume that the condition variable "c" is dependent on an input event. Examining the source code, assume that we wish to move the node x=a+3 above if c. The moved instruction is executed regardless of the control-predicate's outcome; hence the name "speculative transformation."

Carrying out speculative transformations raises three critical issues: variable naming, execution time and safety.

Naming: Consider what would happen if the the transformation shown in Figure 8(C) were performed at the source level. Since `x` would always end up defined as `a+3`, one branch of the conditional would result in an incorrect state. Fortunately, the SSA form of the program ensures that multiply defined source variables – and their corresponding ϕ-functions – maintain the original semantics, regardless of where assignments are moved. Examining Figure 8(C), we see that `x1` and `x2` are defined sequentially. By SSA's naming conventions, the node x3=ϕ(x1,x2) ensures that x3 always carries the assignment corresponding to the original source variable `x`.

Timing: Figure 8 shows how speculative transformations can easily *increase* the execution time of the destination section D. This is not necessarily harmful, since D may in fact possess sufficient slack for both instructions to execute. (Indeed, D may contain an explicit delay.) But if D itself exceeds its own duration constraint, excessive speculative transformations could make matters worse. Thus we take the following approach: the algorithm performs speculative code motion only when feasibility cannot be achieved with the non-speculative variety.

Safety: Perhaps the most critical issue is the correctness of the transformed program. After all, the source code is written by a *human* programmer. When an instruction appears within the body of a conditional (but is free of a transitive dependence on it), one should still assume that the programmer had a good *reason* for putting it there. Often the reason stems from a personal coding style, or perhaps for readability. Also, splitting variable definitions in the style of SSA is a rather unnatural practice at the source level.

Referring back to Figure 7(C), we note that the "eager" execution of "`F`" should be safe if: (1) it contains no observable events, (2) it induces no global side-effects, and (3) it does not cause an exception. We can assume that (1)-(2) hold – otherwise "`F`" would not have been moved in the first place. However, verifying property (3) may be difficult, since there may be an invariant relationship between "`p`" and "`F`" that only the programmer understands.

While this seems to argue against speculative transformation, recall that our objective is to assist programmers in tuning faulty code. And production real-time programmers will find this type of code reordering sadly familiar, since it is usually carried out by hand, and often under the pressure of an approaching release deadline. Our technique can help in this effort, since it helps automate this process by (1) identifying the "good target" instructions to move, (2) by transferring them to their "correct" places, and (3) by analyzing the results. Nonetheless, we do believe that this should be an interactive process (perhaps driven by a graphical front-end), in which the programmer visually checks each transformation.[4]

[4] We discuss this issue in the Conclusion.

For the sake of brevity, however, we present the algorithm in a fully automatic form. Thus we assume that any node n that can be speculatively executed is "pre-checked," and is denoted by the condition $spec(n)$.

Unconditional and Speculative Movability. The preceding discussion leads to three classes of instructions: those that can be unconditionally moved, those that can be moved to execute speculatively, and those which cannot be moved at all. The following definitions distinguish between these cases.

Definition 7.1 (Unconditional Movability) $Mu(\mathrm{S}, V_{bar})$ is the set of nodes in S that do not trigger events, and do not use any variables in V_{bar}; i.e.

$$Mu(\mathrm{S}, V_{bar}) = \{m \in \mathrm{S} \mid m \text{ is not an event } \wedge \; Use(m) \cap V_{bar} = \emptyset\}$$

Then $Umove(n)$ denotes that we can unconditionally schedule node n from S into D:

$$Umove(n) \;\equiv\; DC(n, \mathrm{S}) \subseteq Mu(\mathrm{S}, V_{bar})$$

That is, all of n's data and control dependence ancestors are also unconditionally movable – and when n is moved, they will be moved (or copied) as well. □

Definition 7.2 (Speculative Movability) Additional considerations come into play when a node is speculative executed. Consider the set $Ms(\mathrm{S}, V_{bar})$:

$$Ms(\mathrm{S}, V_{bar}) = \{m \in \; Mu(S, V_{bar}) \mid spec(m) \;\wedge\; \text{if } m \text{ is a } \phi \text{ function then } Umove(m) \text{ holds}\}$$

If a node n is in $Ms(\mathrm{S}, V_{bar})$ then (1) it uses no variables in V_{bar}, (2) it triggers no events, (3) the programmer has checked that it doesn't cause a local exception, and (4) if it is a ϕ function, it can be unconditionally moved (along with its ancestors), which obviates the need for bookkeeping. Then $Smove(n)$ denotes that we can speculatively move node n:

$$Smove(n) \equiv DDC(n, \mathrm{S}) \subseteq Ms(\mathrm{S}, V_{bar})$$

That is, when $Smove(n)$ holds true, all of n's data-dependent ancestors can be moved too, without mandating bookkeeping. □

The Algorithm. The code scheduling algorithm is presented in Figure 7.2. It is composed of three stages: pre-processing, marking/deleting and post-processing. In the pre-processing stage, S's flow graph is traversed in topological order, during which the conditions *Umove* and *Smove* are

```
subroutine Schedule_Section(S, D, d, V_bar)
input: source section S, destination section D, duration constraint d
begin
    foreach node n in S in topological order do
      evaluate Umove(n) and Smove(n);
    make a copy S' of S;
    compute t in S such that wt(t) = max{wt(t') | entry(S) →^{t'} exit(S)};
    while (wt(t) > d)
      call Sched(t, S, S');
      recompute t in S such that wt(t) = max{wt(t') | entry(S) →^{t'} exit(S)};
    end
    delete all unmarked nodes from S';
    delete all predicate nodes guarding null code from S;
    append S' to end of D;
end

subroutine Sched(t, S, S')
begin
    if there is some n ∈ t such that Umove(n) holds then
      begin
          Select first such n ∈ t such that Umove(n) holds;
          foreach node m ∈ DC(n, S) do
            mark[m,S'] := true;
            if m is not a control-predicate then Delete m from S;
          end
      end
    else if there is some n ∈ t such that Smove(n) holds then
      begin
          Select first such n ∈ t such that Smove(n) holds;
          foreach node m ∈ DDC(n, S) do
            mark[m,S'] := true;
            if m is not a control-predicate then Delete m from S;
          end
      end
    else exit("Unable to synthesize.");
end
```

Figure 9: The Section Scheduling Algorithm.

evaluated. A topological traversal ensures that whenever a node n is visited, all ancestor nodes in $DC(n, S)$ have already been processed; hence a single traversal is sufficient to evaluate these conditions. Then a "clone" S' of S is created, which is used to hold the part of the flow graph to be transferred to D.

Next the algorithm searches for a critical trace, and if one exists it invokes subroutine "*Sched.*" *Sched* makes use of array "`mark`," each of whose entries corresponds to a node in S'. Whenever "`mark[`m, S'`] =` *true*," it means that node m will be "moved" into into the destination section. *Sched* examines the critical trace in topological order, looking for node n such that $Umove(n)$ is *true*. If such a node n exists, then closure $DC(n, \mathrm{S})$ is generated; its non-predicate members are deleted from S, while all corresponding nodes in S' are marked. If no unconditionally movable instruction exists, then a transformation of the speculative variety is attempted. And if *no* movable node is present, the program is forced to exit.

At the end, if all critical traces were scheduled, the algorithm proceeds to a post-processing stage. If speculative transformation was carried out then S' will, by definition, contain branching structures with empty predicate nodes. In this case, the nodes on the different branches of the predicate node are merged into a single block.

Finally, S' is attached to the end of the destination section D. Cleaning up, the algorithm deletes control-predicates which guard empty nodes in section S – i.e., the "**if**" nodes whose corresponding bodies and ϕ-functions were completely transferred to S'.

Example, Revisited. We return to our original flight controller example from Figure 5, and subject it to the code scheduling algorithm. The end-result appears in Figure 10. In scheduling S4, "*Sched*" unconditionally moves the function call `compRelAtt`, as well as the other nodes in its dependence closure. This reduces $wt(S4)$ by .25ms, which now stands at 2.57ms. Since $DUR(\mathrm{S4}) =$ 2.25ms), further reductions are made by entering the speculative transformation phase; this results in moving one conditional branch and the function call `safeDtheta` beyond the immovable control-predicate `if (c3)`.

After the transformation, the implementation satisfies the necessary condition for consistency, since the body of S4 requires 2.14ms in the worst-case. Now that $wt(\mathrm{S3}) = 0.68$ms is less than $DUR(\mathrm{S3})$, the code scheduling successfully terminates without further scheduling S3.

In addition to such an instant benefit, the transformation converts possibly wasteful delay into useful computation time, since the new code in S3 can be scheduled within the delay interval between S2 and S4.

```
S6: (S6.start[p]≥p×20ms, S6.finish[p]≤p×20ms+5ms)
    {
S1:   {
        gx1 = φ(gx3, gx0);
        gy1 = φ(gy3, gy0);
        i1 = φ(i3, i0);

        input(GPS, &x1, &y1);                                   [.1ms]
        input(NAV, &theta1);                                    [.1ms]
        input(IMU, &roll1);                                     [.1ms]
      }
S2:   {
        output(Cntrl_out, REQ_VEL);                             [.1ms]
      }
S3:   {
        c1 = GOAL[i1].passed;
        if (c1) {
          gx2 = GOAL[i1].x;
          gy2 = GOAL[i1].y;
          i2 = (i1+1)% NCOORD;
        }
        gx3 = φ(gx2, gx1);
        gy3 = φ(gy2, gy1);
        i3 = φ(i2, i1);

        rtheta1 = compRelAtt(theta1, x1, y1, gx3, gy3);         [.25ms]
        c2 = |rtheta1| < EPS;
        if (c2)
          /* null */
        else
          dtheta3 = safeDtheta(rtheta1, roll1);                 [.43ms]
      }
S4:   (S4.start≥S2.finish+0.75ms, S4.finish≤S2.finish+3.0ms)
      {
        input(Cntrl_in, &vel1);                                 [.1ms]

        if (c2)
          dtheta1 = 0.0;
        else{
          c3 = vel1 < VHIGH;
          if (c3)
            dtheta2 = rtheta1;
          else
            /* null */
        }
        dtheta4 = φ(dtheta1, dtheta2, dtheta3);

        wflap = compFlapw(roll1, vel1, dtheta4);                [.95ms]
        throttle1 = compThrot(roll1, vel1, dtheta4);            [.89ms]
        output(THROT, throttle1);                               [.1ms]
        output(FLAP_Cntrl, wflap);                              [.1ms]
      }
S5:   { /* null */ }
    }
```

Figure 10: Flight Control Program: After Code Scheduling.

8 Concluding Remarks

The TCEL paradigm helps incorporate a higher level of abstraction 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 representation of the designer's intentions, and it need not over-burden the system with unnecessary constraints. Moreover, the event-based semantics enables our scheduling tool to transform the program, and help to resolve conflicts between the timing constraints and the code's actual execution time. Since this is exactly the type of dirty work that compilers do best, a human programmer's time is probably better spent elsewhere.

Practical Considerations. For the sake of brevity we presented the code scheduler in a rather idealized form, abstracting out some of the implementation-related considerations. During our research these factors revealed themselves via three sources: (1) experiences in building a prototype implementation, (2) experiences in dealing with programs significantly larger than the examples in this paper, and (3) discussions with colleagues who design and build production-quality real-time systems. In the following paragraphs we briefly summarize several of these considerations.

Limits of Data-Flow Analysis. The code scheduler heavily relies on contemporary compiler methods, including intra- and inter-procedural data and control analysis. And as with all program transformation algorithms, the limitations of this enabling technology become a constraining factor of our approach. For example, current static data-flow analysis is incapable of disambiguating all pointer aliases (which at worst is an undecidable problem). Thus we cannot always translate the TCEL source into its corresponding "perfect" SSA form. We partially assuage the problem by adopting techniques such as (1) inlining procedures to avoid inter-procedural aliases; (2) rendering in SSA form only those assignments that contain statically analyzable variables; and (3) unrolling loop bodies. Of course these and similar methods will degrade the code scheduler's performance, either by increasing the amount of code, or by decreasing its efficacy. The good news, however, is that dependence analyzers are improving at a rapid rate, and our algorithm will improve along with them.

Limits of Timing Analysis. Another limiting factor is the difficulty of achieving accurate, static timing analysis in the face of more complicated architectures. Quite simply, it has become incredibly difficult to use vendor-supplied benchmarks, and model the interplay between pipelines, hierarchical caches, shared memories, register windows, etc. Thus with an approach like ours, it seems meaningless to predict the execution time of a single instruction (or even a small block). First, the CPU time will probably be too small to make a difference in achieving feasibility, and second, the "noise" in the prediction will be too large.

Thus we have adopted a hierarchical abstraction approach to deal with time predictions. For example, in our flight controller program we accounted only for the CPU-intensive function calls that performed complex operations, while ignoring the execution time of finer-grained instructions. The same approach can be used on larger-grained structures within the *HFG*. Our experience shows the transform engine should usually hunt for the "big-game targets," and forget about the smaller ones.

However after code scheduling is completed, it becomes imperative to verify the result with a more sophisticated timing tool; for example, a good profiler. Performing such re-timing is especially important in a cached memory structure, where code scheduling will always change the instruction alignment. (We note that all modern RISC compilers re-order instructions to some degree; thus the efficacy of *any* source-level timing analysis is diminishing.)

User Interaction. The above two factors argue against a fully automated code synthesis tool. There is also a third factor, which we discussed in reference to speculative code motion. That is, programmers of production-quality, real-time systems will simply not accept a compiler technology that "outsmarts" them, and possibly "disobeys" their intentions. They will, however, enthusiastically embrace a tool that helps tune their systems, but not at the price of sacrificing traceability to their original programs. A simple example illustrates the importance of this. Consider what might happen if an instruction that interacts with the environment fails to be annotated as an event (which could easily happen with memory-mapped IO). If the instruction is relocated outside of its source section, debugging the transformed program could become a nightmare.

All of these considerations argue for a front-end that permits the programmer to interact with the tool during code scheduling. With our scheduling engine as its foundation, a graphical interface would allow a programmer to selectively apply the transformations – and also remain informed of the results.

Pushing Forward. We have recently turned our attention to a more aggressive goal – inter-task transformations to achieve schedulability. In [9] we explore a technique that helps auto-tune an unschedulable *task set* into a schedulable one, by isolating the time-critical threads, and then ensuring that they can be run under a fixed-priority dispatcher.

Finally, we note that the tradition in real-time programs has been to treat all instructions uniformly – as "code" subject to various timing constraints. We have shown that "opening up" a program to consider its event-based semantics can be a great help in achieving feasibility. This approach can potentially be used as a first-line defense in the tuning process, and is usually preferable to measures like hand-optimizing the code, redesigning subsystems, or re-implementing components in silicon.

A Appendix: The Static Single Assignment Form

A program is defined to be in SSA form if each use of a variable is reached by exactly one assignment to it [3]. Thus, a program's SSA representation can be obtained iteratively applying the following process: For each variable in the program, (1) unique names are given to all of its appearances on the left-hand-side of an assignment; and (2) all of the uses reached by that assignment are renamed to correspond to the new name. The following examples demonstrate this process. In the straight line code, each assignment to a variable is given a subscripted name, and all of its uses are then renamed as well.

$$
\begin{array}{lcl}
\texttt{v = f();} & & \texttt{v}_1\texttt{ = f();} \\
\texttt{a = v + 1;} & \Longrightarrow & \texttt{a = v}_1\texttt{ + 1;} \\
\texttt{v = g();} & & \texttt{v}_2\texttt{ = g();} \\
\texttt{b = v + 2;} & & \texttt{b = v}_2\texttt{ + 2;}
\end{array}
$$

Conditional statements require a bit more work in achieving the SSA form. At confluence points in the CFG, merge functions called ϕ-functions are introduced. A ϕ-function for a variable merges its possible values from distinct incoming control flow paths, and produces one argument for each control flow predecessor.

$$
\begin{array}{lcl}
\texttt{if cond} & & \texttt{if cond} \\
\quad\texttt{then v = f();} & & \quad\texttt{then v}_1\texttt{ = f();} \\
\quad\texttt{else v = g();} & \Longrightarrow & \quad\texttt{else v}_2\texttt{ = g();} \\
\texttt{a = v;} & & \texttt{v}_3\texttt{ = }\phi(\texttt{v}_1,\texttt{v}_2)\texttt{;} \\
 & & \texttt{a = v}_3\texttt{;}
\end{array}
$$

Since a ϕ-function is not an actual function to be generated, it is implemented with multiple assignments, as shown below.

$$
\begin{array}{l}
\texttt{if cond} \\
\quad\texttt{then v}_1\texttt{ = f(); v}_3\texttt{= v}_1\texttt{;} \\
\quad\texttt{else v}_2\texttt{ = f(); v}_3\texttt{ = v}_2\texttt{;} \\
\texttt{a := v}_3\texttt{;}
\end{array}
$$

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison Wesley Publishing Company, 1986.

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

[3] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and systems*, 9:319–345, July 1987.

[4] 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.

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

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

[7] M. R. Garey and D. S. Johnson. *Computer and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

[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] P. Gopinath and R. Gupta. Applying compiler techniques to scheduling in real-time systems. In *Proceedings IEEE Real-Time Systems Symposium*, pages 247–256. IEEE Computer Society Press, December 1990.

[11] 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.

[12] S. Hong and R. Gerber. Scheduling with compiler transformations: the TCEL approach. In *Proceedings IEEE Workshop on Real-Time Operating Systems and Software*, pages 80–84. IEEE Computer Society Press, May 1993. *IEEE RTTC Real-Time Newsletter, 9(1/2):80-84.*

[13] Y. Ishikawa, H. Tokuda, and C. W. 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. B. Kenny and K.-J. Lin. Building flexible real-time systems using the Flex language. *IEEE Computer*, pages 70–78, May 1991.

[16] I. Lee, P. Brémond-Grégoire, and R. Gerber. A Process Algebraic Apprach to the Specification and Analysis of Resource-Bound Real-Time Systems. *IEEE Proceedings*, 82(1), January 1994.

[17] 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.

[18] S. Lim, Y. Bae, C. Jang, B. Rhee, S. Min, C. Park, H. Shin, K. Park, and C. Kim. An accurate worst case timing analysis for risc processors. In *Proceedings IEEE Real-Time Systems Symposium*. IEEE Computer Society Press, December 1994. To appear.

[19] 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.

[20] T. Marlowe and S. Masticola. Safe optimization for hard real-time programming. In *Second International Conference on Systems Integration*, pages 438–446, June 1992.

[21] M. Merritt, F. Modungo, and M. Tuttle. Time-Constrained Automata. In *CONCUR '91*, August 1991.

[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] C. Park and A. C. 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.

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

[26] 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.

[27] 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.

[28] M. Younis, T. Marlowe, and A. Stoyenko. Compiler transformations for speculative execution in a real-time system. In *Proceedings IEEE Real-Time Systems Symposium*, 1994. to appear.

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