Slicing Real-Time Programs for Enhanced Schedulability *

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

April 25, 1995

Abstract

In this paper we present an automated, compiler-based technique to help developers synthesize correct real-time systems. The domain we consider is that of multi-programmed real-time applications, in which periodic tasks control a physical systems via interacting with external sensors and actuators. While a system is up and running, these operations must be performed as specified – otherwise the system may fail.

Correctness depends not only on each program individually, but also on complex task interactions which are usually exposed at runtime. Errors at this point are usually remedied by a costly process of instrumentation, measurement and code tuning.

We describe a static alternative to this process, which relies on well-accepted technologies from optimizing compilers and fixed-priority scheduling. Specifically, when an application is found to be overloaded, the scheduling component determines good candidate tasks to get transformed via program slicing. The slicing engine decomposes each of the selected tasks into two fragments: one that is "time-critical," and the other "unobservable." The unobservable part is then spliced to the end of the time-critical code, with the semantics being maintained. The benefit is that the scheduler may postpone the unobservable code beyond its original deadline, which can enhance overall schedulability. While the optimization is completely local, the improvement is realized globally, for the entire task set.

Keywords: Real-time, programming languages, event-based semantics, compiler optimization, program slicing, system-tuning, static priority scheduling, priority assignment.

*This research is supported in part by ONR grant N00014-94-10228, NSF grant CCR-9209333, an NSF Young Investigator Award CCR-9357850. An earlier version of this paper appeared in preliminary form in the *Proceedings of IEEE Real-Time System Symposium*, (December 1993).

1 Introduction

A real-time application is characterized by the existence of two competing factors: its functional specification and its temporal requirements. Functional specifications 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* [8, 18]. 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. Temporal requirements implicitly *limit* the time that can be provided by the system's resources.

Thus, the "art" of real-time development lies in balancing the implementation's resource demands on one hand, and its temporal requirements on the other. If the desired balance cannot be achieved, the result is usually a costly process of low-level system-tuning, involving expensive hardware monitors (e.g., in-circuit emulators, logic analyzers, etc.), taking careful measurements, and then reordering (or restructuring) various key operations. As a last resort, entire subsystems may have to be re-designed altogether.

In this paper we present a static alternative to this process, which is based on two inter-related components: a real-time annotation language called TCEL [11] (for "Time-Constrained Event Language"), and the compiler transformation known as *program slicing* [28, 36, 37]. Surprisingly, while our use of static slicing often leads to longer execution times – and even higher utilizations – it simultaneously helps achieve real-time correctness and schedulability for the entire system. For this reason we call the transformation *real-time task slicing.*

The Language of Time Constrained Events. TCEL's annotation syntax is quite similar to that found in other experimental real-time languages (e.g., [17, 19, 21, 24, 27, 38]). However, the semantics differs significantly, in that it is based on the time-constrained relationships between observable events.

For example, consider a construct such as "**every** 10ms **do** B," where the block of code "B" is executed once every 10ms. The typical approach is to establish timing constraints between *blocks of code*, and in this case *all* of the code in B must fit properly within each 10ms time-frame. But since TCEL's semantics establishes constraints between *externally observable events*, only B's events of interest must fit properly within the 10ms time-frame. This looser semantics yields two immediate benefits. First, the decoupling of timing constraints from code blocks enables a more straightforward implementation of the original, event-based requirements. But more importantly, the unobservable code can be moved to automatically tune the program to its hardware environment.

For simplicity, in the sequel we consider all "**output**" and "**input**" operations to be observable. In practice any relevant instruction can be annotated as event; thus 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.

Real-Time Task Slicing. Consider the following set of requirements, which are typical of many real-time servo-loops.

(1) Every 25ms, read a new measurement from an external sensor.

(2) Using the new sensor reading and the current state, produce an actuator command and update the state.

(3) Send the actuator its output before taking the next sensor reading.

The TCEL program fragment below realizes the specification.

```
L1: every 25ms
      {
L2:   input(Sensor, &data);
L3:   cmd = nextCmd(state, data);
L4:   state = nextState(state, data);
L5:   output(Actuator, cmd);
      }
```

The "**every**" construct on line L1 denotes a statement of cyclic behavior of a positive periodicity. Unlike other real-time languages, the event-based semantics only requires that the observable events L2 and L5 execute within every 25ms, while allowing the executions of local computations L3 and L4 to stretch over the 25ms time frame. Unless the program's control and data dependences are violated, this transformation will always be safe.

The distinction between observable and unobservable operations becomes apparent after the tuning process starts. Specifically, it enables *real-time task slicing* to automatically achieve schedulability. And while the transformation is applied to each individual task, its effect is global, and on the entire system.

The key idea behind this method is based on a simple fact:

> *An application's schedulability improves whenever we can increase the deadlines (or periods) of its constituent tasks.*

The same effect is achieved by allowing a task to slide past its deadline, while maintaining the original event-based semantics. We can realize this benefit by transforming a task, so that its time-sensitive component always executes within its frame, while its unobservable part can be postponed.

To accomplish this, the transform tool decomposes an unschedulable task into two slices: one that is "time-critical" and the other unobservable. Then it "glues" the unobservable slice to the end of time-critical slice, and substitutes the resulting code for the original task.

Figure 1 pictorially illustrates the net effect of this transformation. The downward and upward arrows represent the execution of input and output events within the same time frame.

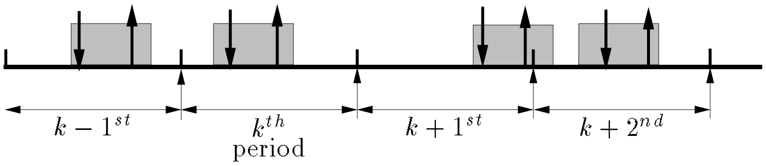

Figure 1: Run-time behavior of a TCEL periodic task.

Although task slicing is applied locally, our application method requires complex interactions between the scheduling analyzer and the task-slicer. Figure 2 illustrates the relationships between the scheduling analyzer/priority assigner, and the real-time task slicer. If the task set is schedulable, the priority algorithm determines the optimal priority assignment for it. If no feasible priority assignment can be found, the scheduling analyzer helps identify the tasks whose transformation will most likely lead to a schedulable system. Then they are fed to slicer, after which the entire task set is again tested for schedulability.

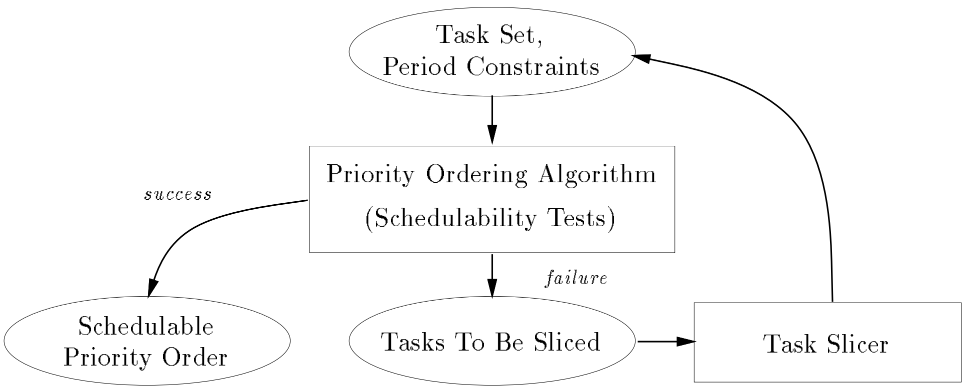

Figure 2: Generalized slicing method for schedulability tuning.

For the sake of presentation, we assume that applications execute on a single CPU. Due to the maturity of uniprocessor scheduling, this makes the analysis somewhat cleaner – and allows us to concentrate on the issue of real-time slicing. But this simplification does not reduce the utility of our approach, since most distributed scheduling algorithms use uniprocessor algorithms at each node[1] – and our method can be adapted as well.

Related Work. Compiler-based techniques for real-time development can conceptually be clustered into two sub-areas – *program transformation* and *timing analysis*.

There has been a growing amount of work within the area of program transformation. The resulting techniques address different problems associated with real-time programming, but they share the goal of enhancing the predictability and schedulability.

In [12] a compiler tool classifies an application program on the basis of its predictability and monotonicity, and creates partitions which have a higher degree of adaptability. Specifically, the tool denotes whether a piece of code belongs in one of four classes; based on this classification, programs are rearranged to help support adaptive run-time scheduling. The objective is to produce a transformed program possessing a smaller variance in its execution time. In [27] a partial evaluator is applied to a source program, which produces residual code that is both more optimized and more deterministic. In [39] an approach to speculative execution is postulated for distributed real-time systems. The idea is that the speculative "shadow threads" are forked off to execute on available resources.

In [11, 15] we show how to use TCEL's event-based semantics to help synthesize feasible task code. (We call a code segment is *infeasible* if its execution time stretches over its specified deadline.) The transformation algorithm corrects such faults via a two-step process. First, the compiler automatically derives a set of timing equations from the language-based timing constraints of a task, so that a real-time scheduler can dispatch the task in an efficient manner. Next, the compiler attempts to correct feasibility faults with respect to the derived dispatch equations. This is done by a variant of Trace Scheduling, in which worst-case paths of the infeasible task are selected, and unobservable code is moved to shorten their execution time.

In this paper we address the more ambitious goal of optimizing multi-threaded applications. But here – unlike in [11, 15] – we have to consider the role of scheduling support. Since scheduling analysis often defines whether timing constraints will be met, the particular scheduling strategy will play a leading role in optimization metric. But schedulers arbitrate between the demands of *several* programs, while a compiler usually works on one program at a time. This traditional separation of concerns between the kernel and the compiler has evolved for many good reasons, and in this paper we strive to maintain it.

[1] For example, a multiprocessor scheduling problem is often modeled as a combination of an allocation sub-problem and multiple instances of single processor scheduling sub-problems [34].

Timing analysis is a key step in building a real-time application; this is especially true in a hard real-time system, in which all deadlines must be met. Many analysis techniques have been proposed, ranging from static, source-based methods to profilers and testing tools, through some combination thereof. While profiling usually produces the tightest results, it presupposes a completely developed system – as well as a test suite that achieves pessimistic worst-case coverage. Static, compiler-based analysis can be used much earlier in the design cycle, and it can usually yield worst-case coverage. But this is also its downside: the result can be a conservative "worst of all worst cases," i.e., an experimentally unachievable measurement. Yet static analysis is developing at a rapid pace, and tools are being produced which can yield tighter results.

The technique reported in [29] is based on a simple source-level timing schema, and it is fairly straightforward to implement in a tool. In [13] another approach for more accurate timing was proposed; the resulting tool was able to analyze micro-instruction streams using machine-description rules, and thus it was retargetable to various architectures. On the other hand, neither approach addresses the problem of predicting architecture-specific timing behavior due to the various latencies inherent in memory hierarchies and pipelines.

New results have begun to account for this timing variance. Zhang *et al.* [40] presented a timing analyzer based on a mathematical model of the pipelined Intel 80C188 processor. This analysis method is able to take into account the overlap between instruction execution and fetching, which is an improvement over schemes where instruction executions are treated individually. In [31] Arnold *et al.* developed a timing prediction method called *static cache simulation* to statically analyze memory and cache reference patterns. A similar but more advanced approach was reported in [23]. While the latter approach is able to predict pipeline stalls as well, both essentially rely on attribute grammars [2] to propagate cache hit information backward in a flow graph.

However, no static timing tool is precise enough to be used with complete confidence for developing production-quality software. Moreover, even sophisticated timing analysis methods such as [23, 31] are not appropriate for fine-grained instruction timing. In Section 3.3 we explain how we can effectively use these tools in spite of the limitations, by also taking advantage of software profiling, as well as static timing prediction. Specifically, our slicing technique does not *require* any static analyzer: it can be used to first transform the program, with the timing carried out later by a runtime profiler.

Remainder of the Paper. The remainder of the paper is organized as follows. In Section 2 we motivate our transformation algorithm via a high-level characterization of discrete-control servo-loops, and we describe some typical scheduling methods used to dispatch them. In Section 3 we provide a technical treatment of program slicing that forms the crux of our transformation. In Section 4 we give an overview of new scheduling methods for the TCEL task model, concentrating on an algorithm that was recently developed for it by researchers at York [5].

In Section 5 we put the analysis method to use – and harness it in our own priority ordering algorithm. The algorithm decides which are the best tasks to get sliced, and determines the resulting priority order for the entire task set. To demonstrate the effectiveness of this algorithm we show the result of a small study we conducted on a task set drawn from an avionics platform.

In Section 6 we describe our prototype implementation, and we conclude the paper in Section 7.

2 Overview of the Approach

As we have seen in Figure 2, "schedulability" is, by definition, a key metric that drives our real-time task slicing, and thus it requires cooperation between the real-time scheduler and the compiler tool. But this presents two competing demands: (1) it is desirable to maintain the traditional separation of concerns between compilation and scheduling; and (2) schedulability depends on complex task interactions that are often exposed at runtime.

In this section we show how real-time task slicing satisfies these demands. In doing so, we discuss the characteristics of a sample target domain – discrete control applications. Since discrete control software possesses many representative properties that can be found in other applications (e.g., multimedia, vision, etc.), this discussion has close analogues in other types of real-time systems.

2.1 Characterization of Discrete Control Software

Many discrete control algorithms possess computations that fit a fixed-rate algorithm paradigm [20], i.e., control-loops which execute repetitively with fixed periods. During each period, the physical world measurement data are sampled, and then actuator commands are computed. Meanwhile, a set of states is updated based on the current state and the sampled data.

The dynamic behavior of a first-order discrete control system can be expressed by the following equations:

$$\begin{aligned} Output_k &= g(State_k, Input_k) \\ State_{k+1} &= h(State_k, Input_k) \end{aligned}$$

In these equations, $Input_k$, $State_k$, and $Output_k$ respectively represent the input, current state, and output of the k^{th} period, while g is an output generation function and h is a state evolution function.

Control equations are thought of as simultaneous relationships (and not as a computation procedure); thus the functions g and h can be implemented in a variety of different ways. The usual practice is to choose a single ordering, and then to code it up as a cyclic control-loop. The actual loop structure is driven by one's personal programming style, or perhaps the availability of generic code modules. But regardless of the choice (unless the underlying control laws are stateless),

g and h mandate key precedence constraints, denoted by "$\prec$":

$$\begin{aligned} Input_k &\prec Output_k \\ State_k &\prec Output_k \\ Input_k &\prec State_{k+1} \\ State_k &\prec State_{k+1} \end{aligned}$$

The typical way to enforce these constraints is to use the "code-based" semantics, and ensure that each iteration of the control-loop completes by the end of its period. This means that the $k+1^{st}$ iteration starts only after the k^{th} iteration ends. Figure 3 illustrates the effect, while the k^{th} iteration is "blown up" in Figure 4.

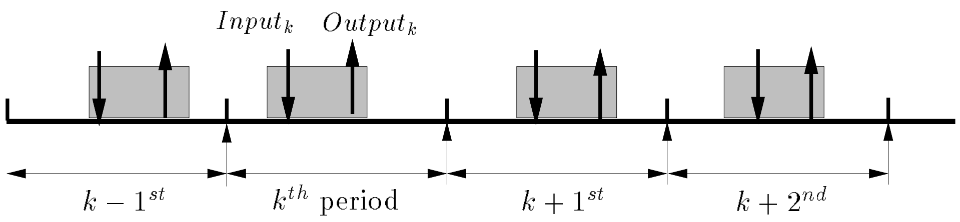

Figure 3: Dynamic behavior of a periodic control-loop.

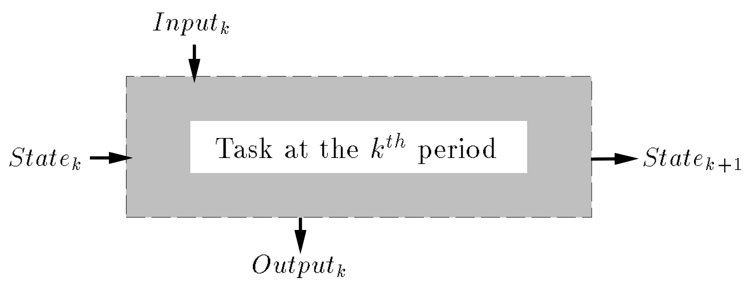

Figure 4: Task instance at the k^{th} period.

2.2 Fixed-Priority Preemptive Scheduling

In any nontrivial system, there are usually many such tasks that share the CPU and other resources. Thus they must be scheduled in a way that allows each of them to adhere to their timing constraints.

This is often done via a fixed-priority, preemptive dispatcher, not only because these schedulers can easily support periodic servo-loops, but also because they possess efficient, offline analysis methods.

Rate-monotonic scheduling, originally developed by Liu and Layland, was the first well-known algorithm of this kind. In their seminal paper [25] they proposed a priority assignment algorithm, in which a task with the shorter period is assigned the higher priority (hence the name rate-monotonic scheduling, or *RMS*). They also showed that such priority assignment is optimal in the sense that whenever it fails to find a feasible priority ordering, neither can any other static priority ordering. However, their algorithm is applicable only to the periodic task model where tasks have fixed periods, deadlines are equal to periods, and tasks are totally independent of each other.

Recent research has made significant enhancements to this model, enhancements which relaxed the original restrictions. In [22] Leung and Merrill showed that a deadline-monotonic priority assignment is also optimal where deadlines may be shorter than periods. In [32] Sha *et al.* presented two protocols which enable tasks to interact via shared resources, while still guaranteeing the tasks' deadlines. Most recently, a group of researchers at the University of York developed a set of analytical techniques which can provide schedulability tests for broad classes of tasks, including those whose deadlines are greater than their periods [4, 35, 33].

In this work we choose the following York notation, mainly due to its generality:

- $\Gamma = \{\tau_1, \tau_2, \ldots, \tau_n\}$ denotes a set of n tasks to be scheduled.
- T_i denotes the period of task τ_i.
- D_i denotes the deadline of task τ_i, relative to the beginning of the current frame.
- c_i denotes the worst-case execution time of τ_i.

If we wish to determine the schedulability of a task τ_i, we use what is called *response time analysis* – defined as the time between when a request for τ_i arrives, and when τ_i finishes its execution servicing the request. If we can confirm that the maximum response time of τ_i is no greater than D_i, we can guarantee that τ_i will meet its deadline even in the worst-case. We first consider the typical "code-based" case, where $D_i \leq T_i$.

Let R_i denote the maximum response time of task τ_i. Then R_i is computed as shown below.

$$R_i = c_i + \sum_{\tau_j \in hp(i)} \lceil \frac{R_i}{T_j} \rceil c_j \qquad \textit{(Eq 1)}$$

where $hp(i)$ is the set of higher priority tasks than τ_i. Observe that R_i is composed of two components, namely execution time c_i and interference. The interference, the second term of *Eq 1*, is the amount of time during which τ_i is preempted by the higher priority tasks in $hp(i)$ since the arrival of its request. As *Eq 1* is a recurrence equation on R_i, an iterative algorithm can compute R_i by initially assigning it c_i, and then generating new values until it converges on a fixpoint (or fails).

But we need a method where deadlines can be longer periods, i.e., where there may be tasks of the form τ_i such that $D_i > T_i$. In this case *Eq 1* is not sufficient, since uncompleted iterations of τ_i can now interfere with the current one. Thanks to [35], the following general equation can be used instead.

$$R_i = \max_{q=0,1,2,\ldots} \{r_{i,q} - q \cdot T_i\} \qquad \textit{(Eq 2)}$$

where

$$r_{i,q} = (q+1)c_i + \sum_{\tau_j \in hp(i)} \lceil \frac{r_{i,q}}{T_j} \rceil c_j$$

```
      every 16ms
      {
L1:     input(Sensor, &data);
L2:     if (!null(data))
        {
L3:       t1 = F1(state);
L4:       t2 = F2(state);
L5:       t3 = F3(data);
L6:       t4 = F4(data);
L7:       state = F5(t1, t2, t3);
L8:       cmd = F6(t1, t3, t4);
L9:       output(Actuator, cmd);
        }
L10:    status_dump("logfile", cmd, state);
      }
```

Figure 5: TCEL program for task τ_2.

Consider the case of three periodic tasks, where the source of task τ_2 is given in Figure 5.

Task	Execution Time	Period	Deadline
τ_1	$c_1 = 400$	$T_1 = 1000$	$D_1 = 1000$
τ_2	$c_2 = 400$	$T_2 = 1600$	$D_2 = 1600$
τ_3	$c_3 = 570$	$T_3 = 2500$	$D_3 = 2500$

Since the periods are equal to the deadlines, rate-monotonic priority assignment is a natural choice. In the above table the row order corresponds to the priority order; *i.e.*, τ_1 is assigned the highest priority. We can carry out the response time analysis for these tasks using *Eq 1* as follows:

For τ_1:	R_1	$=$	400	$<$	$D_1 = 1000$
For τ_2:	R_2	$=$	$400 + \lceil 800/1000 \rceil 400 = 800$	$<$	$D_2 = 1600$
For τ_3:	R_3	$=$	$570 + \lceil 2570/1000 \rceil 400 + \lceil 2570/1600 \rceil 400 = 2570$	$>$	$D_3 = 2500$

Observe that the two high priority tasks τ_1 and τ_2 are schedulable, while τ_3 is not. (R_3 is greater than D_3 when τ_3 runs at priority 3.) In an effort to make the task set schedulable, we might try some hacking: e.g., by promoting τ_3 to the highest priority level. Although this makes τ_3 schedulable, it does not achieve the desired schedulability, since τ_2 will now be unschedulable. Indeed, since the rate-monotonic assignment is optimal, no fixed priority assignment will suffice here – *unless the code-based semantics is abandoned!*

The reason the above task set is unschedulable is obvious: the computation demands of τ_3 exceed the available time. The simulated time line given in Figure 6 pictorially illustrates an unschedulable instance of τ_3.

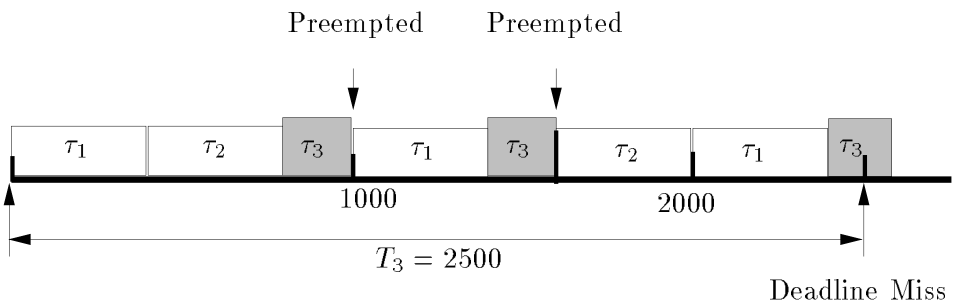

Figure 6: Simulated time line for the example task set.

2.3 Scheduling with Compiler Transformations

When a system is found to be unschedulable, current engineering practice forces programmers to manually pick some critical tasks from the task set, and then to hand-optimize them. Such system-tuning is often repeated many times, until the entire task set finally achieves schedulability. We aim to ease this process by providing a semi-automatic task transformation method, *real-time task slicing.*

Real-time task slicing is based on simple observation:

Traditional real-time scheduling techniques implicitly assume that the entire control-loop *finish by its deadline. But the high-level TCEL semantics mandates that only the observable event-generating operations be finished within the originally specified deadline.*

This observation leads us to the following method. We decompose a task τ into two fragments – one containing all observable event operations, and the other all remaining local operations. We call the former the *IO-hander* and the latter the *state-update* component. We denote them by τ^{IO} and τ^{State}, respectively. Figure 7 demonstrates the decomposition of the control-loop task originally shown in Figure 4.

After the decomposition, we ensure that the IO-handler will execute within its allowable time frame. On the other hand, we may postpone the execution of the state-update part under the worst-case task phasing. Finally, we maintain precedence constraints between τ^{IO} and τ^{State}, originally induced by the task's data and control dependences.

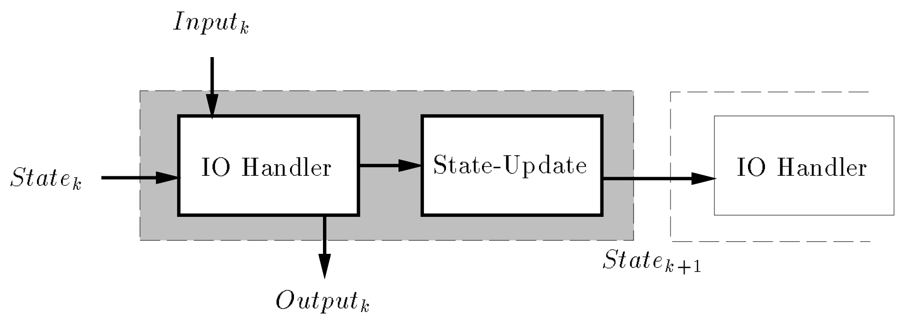

Figure 7: Decomposed task at the k^{th} period.

The task decomposition itself is carried out by static slicing. As we stressed above, we put the greatest emphasis on preserving the timing behavior of observable events and the precedence constraints derived in Subsection 2.1.

But before systematically presenting our slicing procedure, we show its ultimate effect on our example task set. Assume that slicing τ_2 yields the greatest benefit in schedulability. We then decompose τ_2's code into IO-handling τ_2^{IO} and state-update τ_2^{State}, as shown in Figure 8. Their computation times are separately calculated as follows:

$$c_2^{IO} = 2.2\text{ms}, \quad c_2^{State} = 1.9\text{ms}$$

Note that the sum of the two execution times is slightly greater than the original execution time 4.0ms of τ_2. This is due to replication code, additional register loads, etc. – points which we revisit shortly.

```
/* Subtask τ_2^{IO} */
  input(Sensor, &data);
  c = !null(data);
  if (c)
  {
    t1 = F1(state);
    t3 = F3(data);
    t4 = F4(data);
    cmd = F6(t1, t3, t4);
    output(Actuator, cmd);
  }
```

```
/* Subtask τ_2^{State} */
  if (c)
  {
    t2 = F2(state);
    state = F5(t1, t2, t3);
  }
  status_dump("logfile", cmd, state);
```

Figure 8: Two decomposed subtasks of task τ_2.

To enforce that the precedence constraints we splice them together via sequential composition. The net result is the following execution behavior.

$$\tau_2^{IO} \rightarrow \tau_2^{State} \rightarrow \tau_2^{IO} \rightarrow \tau_2^{State} \rightarrow \ldots$$

Our runtime support must guarantee one instance of τ_2^{IO} within each T_2 time-frame; on the other hand, it can let iterations of τ_2^{State} slide between period boundaries.

3 Automatic Task Decomposition by Program Slicing

The idea behind the task decomposition is, as discussed in Subsection 2.3, to accept a task and then slice it into two code components: one containing all observable events, and the other containing only local state-update operations. Many factors make this a difficult compiler problem, among which are intertwined threads of control, nested control structures, complex data dependences between statements, procedure calls in the task code, etc. To concentrate on the issue at hand – i.e. real-time – in this paper we made the following simplifying assumptions.

- Function calls are inlined.
- Loops are finite.
- The programs can be translated into a form in which all false data dependences, such as output dependences and anti-dependences, are eliminated.

The first assumption allows us to avoid *interprocedural* slicing [16]. The next two assumptions simplify the problems induced by false data dependences. Such dependences are caused by variable reuse rather a requirement for data integrity, and real-time slicing will be much more effective if they can be minimized. To do so, we can potentially use a code transformation algorithm such as the rename transformation in [7], the static single assignment (SSA) form translation in [6, 15]. In order to avoid loop-unrolling, we could potentially use dependence-breaking methods, e.g., scalar expansion [3].

However, it is practically impossible to massage a program into a form that is entirely free of false dependences, mainly due to the existence of aliases and pointers. We will get back to this issue shortly, in Section 3.3.

3.1 The Program Slicing Algorithm

Conceptually a *slice* of program P with respect to program point p and expression e consists of P's statements and control predicates that may affect the value of e at point p. We call a pair $\langle p, e\rangle$ a *slicing criterion*, and denote its associated slice by $P/\langle p, e\rangle$. The result is that we can execute the slice $P/\langle p, e\rangle$ to obtain the value of e at location p. Recall our periodic controller task τ_2 of Figure 5. The following fragment is the slice $\tau_2/\langle \text{L9}, \texttt{cmd}\rangle$.

```
L1:  input(Sensor, &data);
L2:  if (!null(data))
     {
L3:     t1 = F1(state);
L5:     t3 = F3(data);
L6:     t4 = F4(data);
L8:     cmd = F6(t1, t3, t4);
L9:     output(Actuator, cmd);
     }
```

Statements L1, L3, L5, L6 and L8 are included in the slice, because variable "`cmd`" depends on their computations (this is called *data dependence*). Also, statement L9 is included because it generates an observable event.[2] Finally, the predicate on line L2 is included, because the execution of statements L3, L5, L6, L8 and L9 (hence the value of "`cmd`") depends on the boolean outcome of the predicate (this is called *control dependence*).

Thus the computation of slices is based on data dependence as well as control dependence. In this regard, using a *program dependence graph* [9, 16, 28] is ideal, since it represents both types of dependences in a single graph. The program dependence graph is defined as follows.

Definition 3.1 The program dependence graph is a directed graph $PDG = (V, E)$, where

[2] We intentionally include L9, as will be discussed in Algorithm 3.1.

- The vertices V represent the task's statements; i.e., assignments, control predicates and observable statements (such as **output** and **input**). In addition there is a distinguished vertex "entry," which represents the root of the task.
- The edges E are of two sorts. An edge $n_1 \xrightarrow{c} n_2$ denotes a control dependence between n_1 and n_2. That is, either (1) n_1 is an entry vertex and n_2 is not nested within any loop or conditional, or (2) n_1 represents a control predicate and n_2 is immediately nested within the loop or conditional whose predicate is represented by n_1. An edge $n_1 \xrightarrow{d} n_2$ denotes a data dependence. That is (1) n_1 defines variable v, and n_2 uses v, *and* (2) control can reach n_2 after n_1 via an execution path along which there is no redefinition of v.

We define "$p \Rightarrow_* q$" to mean that node p can reach node q via zero or more control dependence edges or data dependence edges. □

We assume that the underlying programming language is "perfectly structured." That is, any two statements S_1 and S_2 in the program are in one of the following forms:

1. S_1 is contained in S_2;
2. S_2 is contained in S_1; or
3. S_1 and S_2 are disjoint.

Since many real-time programming languages allow only "structured" programs without unrestricted **goto**s[3], this does not impose serious restrictions on our approach. As a consequence, our definition of control dependence is simpler than that found in [9].

The program dependence graph *PDG* of our controller task τ_2 is shown in Figure 9.

The slice of program P with respect to program point p and expression e (i.e., $P/\langle p, e\rangle$) can be obtained through a traversal of P's program dependence graph. We can extend the definition of a program slice for a set of slicing criteria C in a way that $P/C = \bigcup_{\langle p,e\rangle \in C} P/\langle p, e\rangle$. A simple algorithm to compute the slice is given below. In the algorithm the program point p corresponds to a vertex of *PDG*.

Algorithm 3.1 *Computes the slice $P/\langle p, e\rangle$:*

Step 1 Compute reaching definitions $RD(p, e)$.

Step 2 Compute the slice by a backward traversal of *PDG* such that

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

Figure 10 shows the graph that results from taking a slice of the program dependence graph in Figure 9 with respect to criterion $\langle \mathrm{L9}, \texttt{cmd}\rangle$.

[3] We disallow **break** statement as well, since it is a special instance of **goto**.

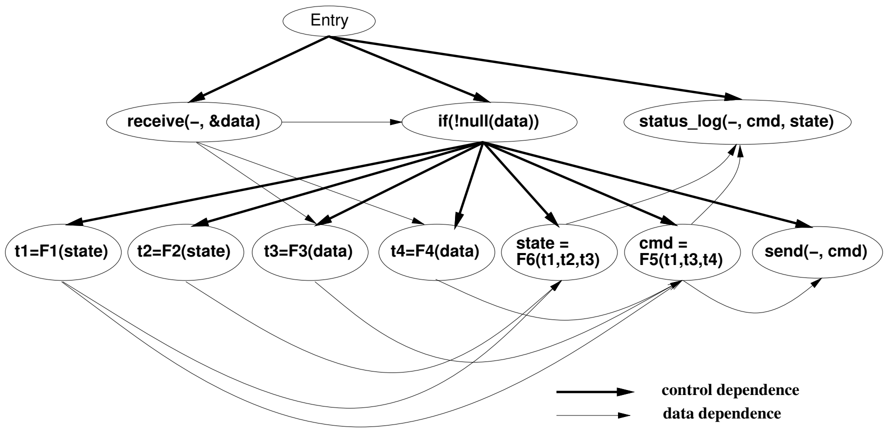

Figure 9: Program dependence graph.

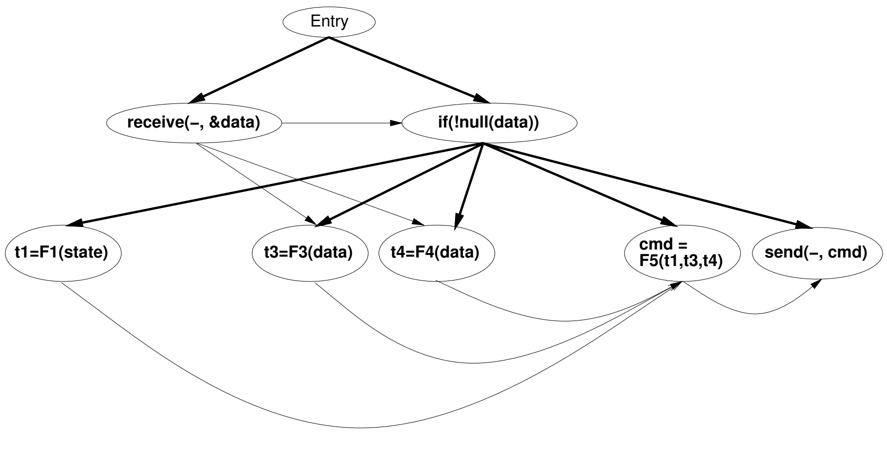

Figure 10: Slice with respect to criterion ⟨L9, `cmd`⟩.

One of the essential points in using our task decomposition algorithm is to providing right slicing criteria for the algorithm, so that the computed I/O slice of a task "covers" all the observable behaviors of the original task. Criteria selection can be automated by means of the observable event specification, or it can be manually performed by way of graphical user interface.

Let $C_{IO}(\tau)$ be a set of slicing criteria for I/O slice of task τ. Then the task decomposition algorithm is given below:

Algorithm 3.2 *Decompose task τ into τ^{IO} and τ^{State}:*

- **Step 1** Compute the slice of τ with respect to $C_{IO}(\tau)$ using Algorithm 3.1. The generated slice $\tau/C_{IO}(\tau)$ becomes τ^{IO}.
- **Step 2** Delete from τ all repeated statements of τ^{IO} except for the conditional statements. The remaining code becomes τ^{State}.

Figure 8 shows the two subtasks τ_2^{IO} and τ_2^{State} of τ_2 computed by Algorithm 3.2 with slicing criteria $C_{IO}(\tau) = \{\langle L1, data\rangle, \langle L9, cmd\rangle\}$.

3.2 Slicing, Splicing and Timing Analysis

Program slicing may easily increase worst-case execution times of tasks for a number of reasons: (1) control structures are replicated and will be executed twice; (2) splitting a basic block may increase the number of register load and store operations [2]; and (3) worst-case execution time paths of the two resultant subtasks may be incorrectly derived. We take a close look at the last factor, since it tends to take up the greatest portion of the increase, though it is mainly an artifact of overly-conservative timing prediction.

After a conditional of a task is sliced and then spliced, we must carefully correlate the duplicated condition predicates. If not, our static analysis will give us a wildly conservative worst-case execution time. Figure 11 pictorially depicts this case. The original task τ consists of one conditional, one branch of which is IO-generating code "`IO`," and the other is state-update code "`ST`." The predicted worst-case execution time of τ is:

$$wt(\tau) = wt(\mathtt{c}) + \max\{wt(\mathtt{IO}), wt(\mathtt{ST})\}.$$

In Figure 11 τ is sliced into two subtasks τ^{IO} and τ^{State}. Their worst-case execution times are also given below.

$$\begin{aligned} wt(\tau^{IO}) &= wt(\mathtt{c}) + wt(\mathtt{IO}) \\ wt(\tau^{State}) &= wt(\mathtt{c}) + wt(\mathtt{ST}). \end{aligned}$$

Consequently, the worst case execution time of the transformed task τ' ($\equiv \tau^{IO}; \tau^{State}$) may be measured as:

$$wt(\tau') = 2 \cdot wt(\mathtt{c}) + wt(\mathtt{IO}) + wt(\mathtt{ST}),$$

which is much larger than $wt(\tau)$.

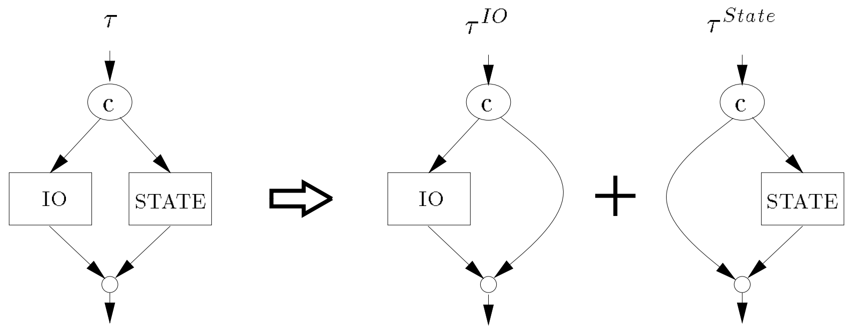

Figure 11: Slicing a conditional.

However, tighter worst-case execution time can be easily obtained by correlating the conditional predicate of the subtask τ^{IO} with that of the subtask τ^{State}. For example, if "`IO`" (in Figure 11) is executed in the subtask τ^{IO}, then we know that the empty left branch will be executed in subtask τ^{State}. Thus $wt(\tau')$ can be refined as follows:

$$wt(\tau') = 2 \cdot wt(\texttt{c}) + \max\{wt(\texttt{IO}), wt(\texttt{ST})\}$$

For the given two subtasks of τ_i, we carry out the following simple steps which is based on the notion of predicate correlation to compute tight worst-case execution times of subtasks τ_i^{IO} and τ_i^{State}.

Step 1 Calculate c_i' by running a timing tool with the code of τ_i'.

Step 2 Calculate c_i^{IO} by running a timing tool with the code of τ_i^{IO}.

Step 3 Calculate c_i^{State} such that $c_i^{State} = c_i' - c_i^{IO}$.

This will serve as a good rough estimate for the transformed task code. Then we can use a profiler to account for the two other factors that incur timing overhead, as we discuss in the next subsection.

3.3 Practical Considerations

As we have stated, the above presentation of real-time slicing is rather idealized; we abstracted out some of the practical considerations that would have confused our focus. On the other hand, our success is contingent upon the inherent limitations of static program analysis. We faced in limitations when building our prototype slicer tool (Section 6), which uses an existing dependence analyzer, as well as a course-grained static timing analyzer. In the following discussion, we elaborate on some of these problems, and point out some ways of working around them.

In the preceeding subsection, we assumed that programs are rendered in a form where all false dependences are effectively removed. Fortunately the correctness of our approach does not hinge on this assumption, which is a good thing – since static data-flow analysis is incapable of disambiguating all pointer aliases (at worst an undecidable problem). The abovementioned techniques of inlining and loop unrolling partially assuage this problem; inlining can obviate doing complex, interprocedural analysis, while unrolling (and the associated renaming) can help expose output and anti-dependences. Of course these and similar methods can dramatically increase code size, which will in turn lead to more pressure on memory. The associated trade-off analysis is best made by the developers of the system.

But for the sake of soundness, we have to be conservative. That is, in practice we only remove false dependences between those assignments that contain statically analyzable variables, and treat remaining "dependences" as true, flow dependences. At worst, we end up with code that *appears* totally unsliceable – when it may, in fact, be amenable to slicing. Again, the developers can be of enormous help, by manually breaking some of the false dependences.

We also note that dependence analyzers are improving at a fast rate (see [14, 30]), and our algorithm will improve along with them. For example, if we incorporate the recent advances in loop dependence analyses such as those in the *Omega Test* [30], we may not have to unroll loops to slice a real-time task. Moreover, we can obtain better slices for loops using techniques like *loop distribution*.

We also rely on achieving reasonable execution time bounds for the code segments. But in the face of more complicated architectures, getting tight, static timing bounds is getting more difficult – due to the interplay between pipelines, hierarchical caches, shared memories, register windows, etc.

Thus we have adopted a two-tier approach to deal with time predictions. We make use of a timing analyzer for a rough, initial estimate. Then after program slicing is completed, we verify the result with a more sophisticated profiling tool, that actually runs the program. Performing such re-timing is especially important in a cached memory structure, where code scheduling will always change the instruction alignment.

4 Scheduling the Sliced Tasks

Now assume we have a set of n tasks Γ, numbered τ_1 to τ_n. If some of the tasks are sliced, how do we best schedule Γ to ensure that the event-based semantics are maintained? Any such scheme is dependent on two elements:

(1) A scheduling policy that can exploit our task model; i.e. while the τ^{State} components can miss their original deadlines, the precedence constraints between instances τ^{IO} and τ^{State} must be maintained.

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

In [10] we present a RMS-based method, in which each process receives *two* priorities, one for τ^{IO} and one for τ^{State}. Its principal strength is a simple dual-priority assignment rule, and an efficient analysis test to determine schedulability. Its weakness is that the online component lacks the simplicity found in pure, static priority scheduling. The dual-priority scheme mandates a dynamic priority-exchange mechanism, which in turn requires additional kernel support.

So the following question arises: when can a set of transformed TCEL tasks be scheduled under a fully preemptive, static priority scheme? Burns [5] 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 [25]. Thus the rate-monotonic priority assignment may not be the optimal one.

Given set Γ' of transformed TCEL tasks

$$\begin{aligned}\tau_1' &= \tau_1^{IO}; \tau_1^{State}\\ \tau_2' &= \tau_2^{IO}; \tau_2^{State}\\ &\vdots\\ \tau_n' &= \tau_n^{IO}; \tau_n^{State}\end{aligned}$$

it turns out the appropriate priority assignment is not only dependent on the deadlines (as in the pure deadline-monotonic model), but also on the respective execution times of each IO-handler and state-update component. In [5] 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 the programmer may have to go back to the system design step and consider more aggressive system-tuning.

For given task set $\Gamma = \{\tau_1, \tau_2, \ldots, \tau_n\}$ Burns' priority assignment algorithm accepts the pre-processed task set $\Gamma' = \{\tau_1', \tau_2', \ldots, \tau_n'\}$ as its input, where τ_i' is a sliced version of τ_i. It then begins looking for a task that can run at the lowest priority (level n)[4]. After such a task, say, τ_k' is found, the algorithm proceeds to search the new task set $\Gamma' - \{\tau_k'\}$ for the second lowest priority task, and so on. There is an important fact that leads to the optimality of this algorithm: while a task is being tested for priority level p, all $p-1$ tasks whose priorities have not yet been assigned are

[4] For n tasks, n denotes the lowest priority level, and 1 the highest.

assumed to run at higher priorities. In fixed-priority preemptive scheduling, since a lower priority task can never preempt the higher priority tasks, selections made for priority levels p or below will not affect those above p.

During this priority ordering, the schedulability test of τ_i' ($\equiv \tau_i^{IO}; \tau_i^{State}$) for priority p yields the following two conditions:

(1) Whether τ_i^{IO} can always run within time D_i at priority p, and

(2) Whether there exists some integer q such that q consecutive iterations of τ_i' can run within $q \cdot T_i + D_i$ at priority p where $q \geq 1$.

Condition (1) is required by the TCEL's semantics; condition (2) accounts for the case where at least one iteration of τ_i^{State} is delayed. The schedulability test boils down to a check to see if the maximum response time of τ_i^{IO} is no greater than D_i in either case.

The maximum response time (denoted by R_i^{IO}) of τ_i^{IO} with respect to $hp(i)$ is computed as below:

$$r_{i,q} = q(c_i^{IO} + c_i^{State}) + c_i^{IO} + \sum_{\tau_j \in hp(i)} \lceil \frac{r_{i,q}}{T_j} \rceil c_j$$

$$R_i^{IO} = \max_{q=0,1,2,\ldots} \{ r_{i,q} - q \cdot T_i \} \qquad \textit{(Eq 3)}$$

We must subtract $q \cdot T_i$ from $r_{i,q}$ to obtain the real response time, since $r_{i,q}$ is measured from the start of the q^{th} period prior to the current period. Although q is denoted as an unbounded number in *Eq* 3, it can be trivially shown that there exists bounded response time R_i^{IO} as long as utilization of $hp(i) \cup \{\tau_i\}$ is less than 100% [35].

On the other hand, q is bounded below by $\frac{r'_{i,q}}{T_i + D_i}$ where

$$r'_{i,q} = (q+1)(c_i^{IO} + c_i^{State}) + \sum_{\tau_j \in hp(i)} \lceil \frac{r'_{i,q}}{T_j} \rceil c_j.$$

The intuition behind the equation is that the execution timeline of τ_i' repeats only after τ_i' gets to complete within its proper time-frame.

Now recall the unschedulable task set we showed in Subsection 2.2. Suppose that only τ_2 was sliced. This requires priority rearrangement among the tasks, since RMS is no longer optimal in the transformed task model. The result of new priority ordering is as follows:

$$\tau_3 \prec \tau_1 \prec \tau_2'$$

In the next section we show how this ordering is obtained. But given that we have an ordering, we can check it using *Eq 3*.

For τ_3:				
R_3	$=$	570	$<$	$D_3 = 2500$
For τ_1:				
R_1	$=$	$400 + \lceil 970/2500 \rceil 570 = 970$	$<$	$D_1 = 1000$
For τ_2':				
$r'_{2,1}$	$=$	$2 \cdot 410 + \lceil 2750/2500 \rceil 570 + \lceil 2750/1000 \rceil 400 = 2750$	$<$	$T_2 + D_2 = 3200$
$r_{2,0}$	$=$	$220 + \lceil 1590/2500 \rceil 570 + \lceil 1590/1000 \rceil 400 = 1590$		
$r_{2,1}$	$=$	$410 + 220 + \lceil 2400/2500 \rceil 570 + \lceil 2400/1000 \rceil 400 = 2400$		
R_2^{IO}	$=$	$\max\{1590, 2400 - 1600\} = 1590$	$<$	$D_2 = 1600$

As a result, the task set is shown to be schedulable under the new priority assignment.

5 Priority Ordering with Task Slicing

In this section we present the missing link; i.e., the algorithm that determines which tasks to slice, and which to leave intact. The priority assignment algorithm in [5] expects that all tasks in Γ are sliced before they are submitted for priority assignment. However, it is typically not desirable to slice all tasks in the application due to execution time overhead incurred. As an example, consider a task set whose utilization is 0.96. Suppose that task slicing increases the worst-case execution times of most tasks by 5%. If we naively slice them all, this will result in utilization of 1.008, and will render the task set permanently unschedulable. Moreover, since we view slicing as a means of tuning an application, it should selectively be applied to tasks which will realize the greatest benefit.

To address this problem, we present an algorithm that not only finds a feasible task priority ordering, but also picks only a small subset of tasks to slice. For a given ordered list of tasks $\Gamma = [\tau_1, \tau_2, \ldots, \tau_n]$, we make the following definitions.

- sliced(τ_i) : a boolean variable denoting whether or not τ_i is sliced.
- $c_i' = c_i^{IO} + c_i^{State}$.

We refer to a certain permutation Γ' of Γ as a *configuration*, i.e. Γ' denotes a priority ordering[5] of the tasks in Γ, and sliced(τ_i) is defined for all $\tau_i \in \Gamma'$. There are $n!$ different priority orderings, and 2^n possible slicing choices. Thus the algorithm's job is to choose a task configuration among $2^n \cdot n!$ distinct ones in an efficient manner.

[5]The first task in the list has the highest priority.

Definition 5.1 (Feasibility) For a given Γ, a configuration Γ' is said to be *feasible iff* all tasks in Γ' meet their deadlines under the priority ordering and slicing choice denoted by Γ'.

5.1 Feasibility Test

Our problem is to slice for schedulability, which is complicated by the fact that for a task τ_i, $c_i \neq c_i^{IO} + c_i^{State}$. Thus it seems inevitable to search the entire solution space of size $2^n \cdot n!$ in order to find a feasible task configuration.

Fortunately, there are cases where we can make a slicing decision without exhaustively exploring the search space. We rely on the response time analysis summarized by equations *Eq 2* and *Eq 3* to find these cases. To be specific, we make use of the following schedulability test.

$$
\begin{array}{l}
\textit{Feasible}(\mathcal{L}, \tau_k) \equiv \\
\quad \textbf{if } \neg\text{sliced}(\tau_k) \textbf{ then } \max_{q=0,1,2\ldots}\{r_{k,q} - q \cdot T_k\} \leq D_k \\
\quad \textbf{else } \max_{q=0,1,2\ldots}\{r'_{k,q} - q \cdot T_k\} \leq D_k \\
\text{where} \\
\quad S = \{\tau_i \in \mathcal{L} \mid \text{sliced}(\tau_i)\}, \\
\quad r_{k,q} = (q+1)c_k + \sum_{\tau_j \in \mathcal{L}-S} \lceil \frac{r_{k,q}}{T_j} \rceil c_j + \sum_{\tau_j \in S} \lceil \frac{r_{k,q}}{T_j} \rceil c'_j, \text{ and} \\
\quad r'_{k,q} = q \cdot c'_k + c_k^{IO} + \sum_{\tau_j \in \mathcal{L}-S} \lceil \frac{r'_{k,q}}{T_j} \rceil c_j + \sum_{\tau_j \in S} \lceil \frac{r'_{k,q}}{T_j} \rceil c'_j.
\end{array}
$$

The proposition "$\neg\text{sliced}(\tau_k) \wedge \textit{Feasible}(\mathcal{L}, \tau_k)$" denotes that the unsliced τ_k is schedulable with tasks in $\mathcal{L}$ running at higher priorities. Similarly, "$\text{sliced}(\tau_k) \wedge \textit{Feasible}(\mathcal{L}, \tau_k)$" means that τ_k, after getting sliced, is schedulable with tasks in $\mathcal{L}$.

5.2 The Algorithm

We now present the selection algorithm, which uses the following variables:

- $\Gamma = [\tau_1, \tau_2, \ldots, \tau_n]$: The input task list, initially ordered in nondecreasing order of the deadlines. Such a deadline monotonic ordering is desirable as a starting point, since most tasks, except for a small number of tasks to be sliced will end up consistent with it.

- Parameters $\mathcal{L}_1$ and $\mathcal{L}_2$: $\mathcal{L}_1 @ \mathcal{L}_2$ holds the current list of tasks, where "@" denotes the *list append* operation. In every invocation, *Search*$(\mathcal{L}_1, \mathcal{L}_2)$ returns either the priority-ordered list of the tasks, or *false* if it cannot find any feasible ordering among them.

In every invocation, *Search* attempts to assign the last task in $\mathcal{L}_1$ (variable "τ" in Figure 12) the priority level $|\mathcal{L}_1| + |\mathcal{L}_2|$. The condition on line (1) denotes that the algorithm has already generated a complete task configuration of Γ.

The condition on line (2) means that the algorithm has checked all tasks in lists $\mathcal{L}_1$ and $\mathcal{L}_2$ for priority level $|\mathcal{L}_1| + |\mathcal{L}_2|$, but it can assign none of them that priority. Thus *false* is returned.

Otherwise, the algorithm attempts to find a feasible priority assignment for tasks up to the current priority (line (4)). If this cannot be done without τ, then τ is infeasible at the current priority. In this case the tail recursion is invoked on line (12), which will attempt to find another ordering.

But when the higher priority tasks are schedulable, τ is checked for feasibility with them. If so, a new ordering $L@[\tau]$ is returned. Otherwise, the algorithm slices τ, and sees if τ' is feasible, after which it returns $L@[\tau']$.

In spite of its worst-case complexity, the algorithm computes results very fast in almost all interesting cases. The reason is because it attempts to first assign tasks deadline monotonic priorities, and most of the tasks end up having priorities consistent with this order. We also note that a task is only sliced *on demand*, when its unsliced version cannot be scheduled within the current configuration.

```
        algorithm PriAssign(Γ)
        begin
          return(Search(Γ, []));
        end

        list function Search(L1, L2)
        case
(1)       when L1 = L2 = []: return([]);
(2)       when L1 = [], L2 ≠ []: return(false);
(3)       when L1 = L1'@[τ]:
(4)            L = Search(L1'@L2, []);
(5)            if L ≠ false then
(6)              if Feasible(L, τ) then
(7)                return(L@[τ]);
(8)              else
(9)                τ' = Slice(τ);
(10)               if Feasible(L, τ') then
(11)                 return(L@[τ']);
                 end
               end
(12)           return(Search(L1', [τ]@L2));
        end
```

Figure 12: Algorithm for priority ordering with slicing decision.

5.3 A Larger Example

In a somewhat larger example, we adapted the periodic tasks described in [26, 35], factoring in display server activity, as well as the IO and state-update components (while modifying the execution times accordingly). The adapted task set had a utilization of 0.836, and it was unschedulable under any static priority ordering. The resultant timing specification of our task set is given in Table 1, where the time unit is 1 microsecond.

We make the following assumptions for the task set, which we have found to be representative.

1. Only small portion of a task – no more than 25 % of the original task code in terms of the worst case execution time – can be sliced.

2. Slicing incurs no more that 5 % increase in a task's worst-case execution time.

	T	D	c	c^{IO}	c^{State}
τ_1	1000	1000	51	51	0
τ_2	25000	5000	2000	1600	500
τ_3	25000	5000	1000	800	250
τ_4	40000	5000	2000	1600	500
τ_5	50000	20000	3000	2400	750
τ_6	200000	20000	3000	2400	750
τ_7	50000	25000	5000	4000	1250
τ_8	59000	25000	8000	6400	2000
τ_9	80000	80000	9000	7200	2250
τ_{10}	80000	80000	2000	1600	500
τ_{11}	100000	80000	8000	6400	2000
τ_{12}	100000	100000	5000	4000	1250
τ_{13}	200000	100000	3000	2400	750
τ_{14}	200000	100000	1000	800	250
τ_{15}	200000	120000	1000	800	250
τ_{16}	200000	140000	2000	1600	500
τ_{17}	1000000	1000000	1000	800	250
τ_{18}	1000000	1000000	1000	800	250

Table 1: Example task set.

The priority ordering algorithm chose to slice tasks τ_4, τ_7 and τ_{16}, thereby making the task set schedulable. The utilization grew slightly to 0.844. We show the result in Table 2, where "R" denotes the maximum response time of an unsliced task, and R^{IO} and R^{State} respectively represent the maximum response times of the two components of an sliced task.

6 TimeWare/SLICE: the Prototype Implementation

Our prototype tool is called TimeWare/SLICE, and it harnesses the "*Search*" algorithm to determine which tasks should be sliced. However, the programmers get to graphically pick the slicing criteria, and they can "veto" any slicing decision.

	T	D	c	c'	R	R^{IO}	R^{State}	Sliced?
τ_1	1000	1000	51	51	51	0	0	n
τ_2	25000	5000	2000	2100	2153	0	0	n
τ_3	25000	5000	1000	1050	3204	0	0	n
τ_4	40000	5000	2000	2100	0	4855	5406	y
τ_6	200000	20000	3000	3150	8559	0	0	n
τ_5	50000	20000	3000	3150	11712	0	0	n
τ_8	59000	25000	8000	8400	20171	0	0	n
τ_7	50000	25000	5000	5250	0	24375	28829	y
τ_9	80000	80000	9000	9450	38339	0	0	n
τ_{10}	80000	80000	2000	2100	42643	0	0	n
τ_{11}	100000	80000	8000	8400	71372	0	0	n
τ_{12}	100000	100000	5000	5250	79780	0	0	n
τ_{13}	200000	100000	3000	3150	96747	0	0	n
τ_{14}	200000	100000	1000	1050	97798	0	0	n
τ_{15}	200000	120000	1000	1050	98849	0	0	n
τ_{16}	200000	140000	2000	2100	0	139890	140441	y
τ_{17}	1000000	1000000	1000	1050	141492	0	0	n
τ_{18}	1000000	1000000	1000	1050	142543	0	0	n

Table 2: Priority assignment with program slicing.

For a variety of reasons, we have found this semi-automated strategy to best fit our problem domain. First, users can help tighten both the automatic dependence analysis, by using assertions to manually break false dependences that our static analyzer cannot handle (e.g., most pointer aliases). This results in better slices, and greater possibility of achieving schedulability.

In a similar manner, the users can tighten the timing analysis by running their selected programs through special measurement tools – and not the generic timing analyzer. Moreover, enhanced dependence analysis will also increase the precision of timing estimates, since it will enable correlating more conditional blocks with aliased controlling predicates (see Section 3.2).

But the most important reason to provide intensive user interaction is to maintain traceability to the original source code. This argues for a front-end that permits the programmer to interact with the tool during each step of system-tuning. With our transformation engine as its foundation, a graphical interface allows a programmer to selectively apply the transformations – and also remain informed of the results.

TimeWare/SLICE Functionality. The key features of the tool are as follows:

- It computes a program slice with respect to a given slicing criterion.
- It present the code at the source-level.
- It allows users to save a computed slice, and to carry out operations between the current and the saved slices. These operations include intersection between slices, union of slices and subtracting one slice from another. These commands are used to segregate special threads from given task code.

• It automatically generates the transformed task code.

TimeWare/SLICE Tool Screens. A source program is displayed on the two tool windows: one called the *primary window* and the other the *scratch-pad window*. Figure 13 demonstrates a possible layout on the tool screen of a SUN Sparc station equipped with a 17-inch display.

The primary window provides users with a workplace where they can pick a slicing criterion and get the slicing result. The result is shown as a set of highlighted source lines on the window. The scratch-pad window provides with a buffer space where a user can temporarily store a pre-computed slice. When a user carries out operations between two slices (one on the primary window and the other on the scratch-pad window), the primary window works as if it were an accumulator. That is, the primary window provides the first operand and gets the result.

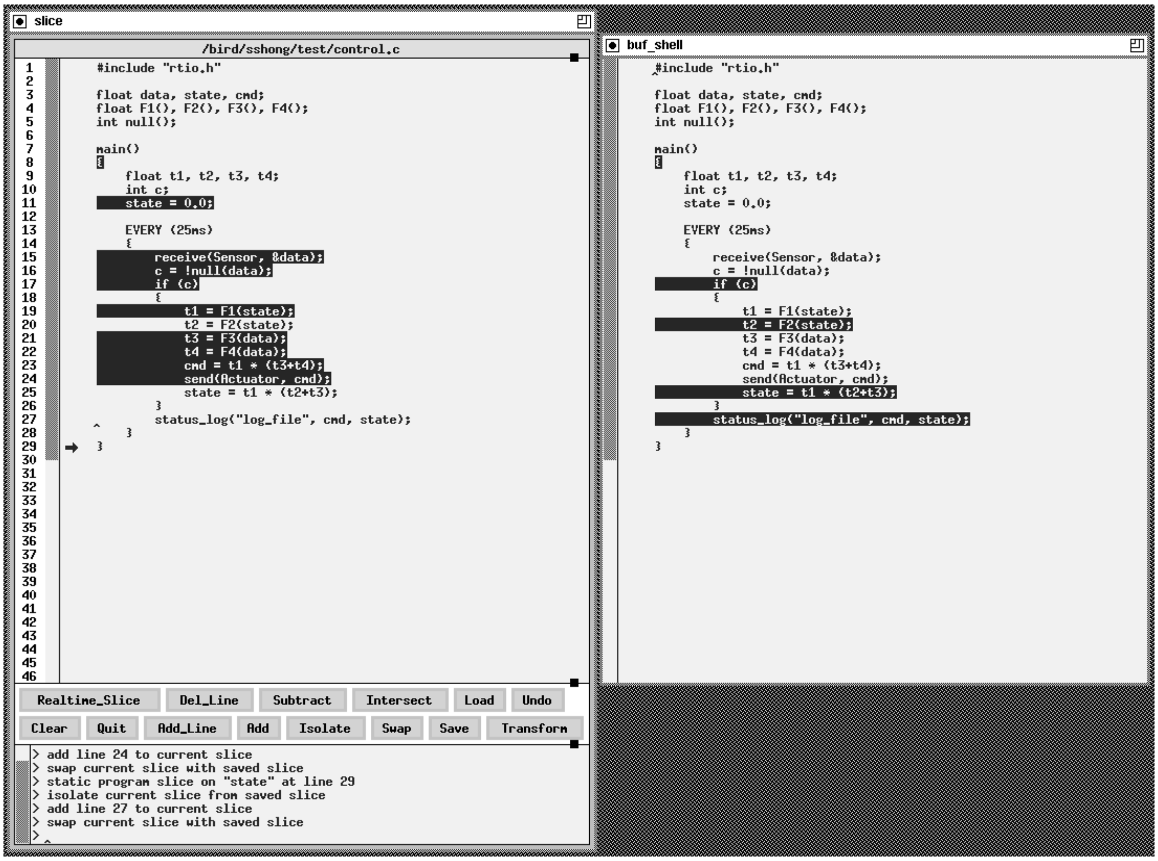

Figure 13: Tool screen of TimeWare/SLICE.

Figure 13 shows the tool's display of the two slices in our example, where we have substituted F5 and F6 with simple arithmetic operations. Figure 14 shows the transformed code that was generated by splicing the two slices together.

```
#include "rtio.h"

float data, state, cmd;
float F1(), F2(), F3(), F4();
int null();

main()
{
    float t1, t2, t3, t4;
    int c;
    state = 0.0;

    EVERY (25ms)
    {
        receive(Sensor, &data);
        c = !null(data);
        if (c)
        {
            t1 = F1(state);
            t3 = F3(data);
            t4 = F4(data);
            cmd = t1 * (t3+t4);
            send(Actuator, cmd);
        }

        if (c)
        {
            t2 = F2(state);
            state = t1 * (t2+t3);
        }
        status_log("log_file", cmd, state);
    }
}
```

Figure 14: Transformed code.

Implementation. The prototype implementation of TimeWare/SLICE is based on a dynamic program slicing tool SPYDER developed at the Purdue University [1]. SPYDER is a program debugging tool relying on dynamic slicing, and it consists of two components: a modified version of GCC (GNU C compiler) and GDB (GNU symbolic debugger). The role of the modified GCC is to produce the program dependence graph for an input program as well as the object code. SPYDER traverses the graph to compute a static program slice.

We had to tailor the implementation of SPYDER due to the following limitations.

(1) It does not allow users to pick a general slicing criterion. Instead, it limits the criterion to a variable name.

(2) SPYDER is a program analysis tool, whereas TimeWare/SLICE actually transforms the program text.

(3) SPYDER's static slicer is not complete, which results in incorrect slices being produced. For example, the static data flow analyzer of the modified GCC does not detect redefinitions of a global variable within a function, and SPYDER cannot account for this limitation. We had

to retool TimeWare/SLICE to conquer this problem.

As we described in Section 3.3, our approach to dependence analysis is to be conservative – by assuming that every function call has a potential to redefine every global variable. SPYDER's interprocedural analysis is limited, and its goal allows it to take an alternative, "optimistic" approach. While our assumptions may result in too large an IO slice, users of TimeWare/SLICE can still modify it using the editing facilities. But in such a case it is always better to be safe than sorry.

As a final note, we point out that the implementation of TimeWare/SLICE enjoys all the benefits of GNU-based software, the most prominent being its portability to various hardware platforms.

7 Conclusion

In this paper we presented an automated system-tuning approach for fixed-priority, preemptive real-time systems. The approach consists of three interrelated components, namely, a real-time programming language TCEL, a task slicing algorithm, and a new fixed-priority scheduling strategy.

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. Most importantly, it enables our compiler tools to transform the program.

For the underlying scheduling paradigm, we have concentrated on rate-based scheduling, since it is one of the best understood areas in the real-time literature, and one of the most widely embraced methods by practitioners. Unfortunately, the tradition is disappointing in that one always considers 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.

References

[1] H. Agrawal, R. DeMillo, and E. Spafford. Debugging with dynamic slicing and backtracking. *Software Practice and Experience*, 23(6):590–616, June 1993.

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

[3] F. Allen, B. Rosen, and K. Zadeck. *the forthcoming Optimization in Compilers.* Addison Wesley Publishing Company, 1992.

[4] N. Audsley. Optimal priority assignment and feasibility of static priority tasks with arbitrary start times. Technical Report YCS 164, Department of Computer Science, University of York, England, December 1991.

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

[6] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13:451–490, October 1991.

[7] R. Cytron, A. Lowry, and K. Zadeck. Code motion of control structures in high-level languages. In *Conference Record 13th Annual ACM Symposium on Principles of Programming Languages*, pages 70–85. ACM Press, January 1986.

[8] 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.

[9] 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.

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

[11] R. Gerber and S. Hong. Compiling real-time programs with timing constraint refinement and structural code motion. *IEEE Transactions on Software Engineering*, May 1995. To Appear.

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

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

[14] L. Hendren, J. Hummel, and A. Nicolau. Abstractions for recursive pointer data structures: Improving the analysis and transformation of imperative programs. In *Proceedings of the ACM*

SIGPLAN Conference on Programming Language Design and Implementation, pages 249–260. ACM Press, June 1992.

[15] 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.*

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

[17] 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.

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

[19] 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.

[20] 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.

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

[22] J. Leung and M. Merill. A note on the preemptive scheduling of periodic, real-time tasks. *Information Processing Letters*, 11(3):115–118, November 1980.

[23] 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 of IEEE Real-Time Systems Symposium*, pages 97–108. IEEE Computer Society Press, December 1994.

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

[25] 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.

[26] C. Locke, D. Vogel, and T Mesler. Building a predictable avionics platform in ada: A case study. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 181–189. IEEE Computer Society Press, December 1991.

[27] 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.

[28] 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.

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

[30] W. Pugh and D. Wonnacott. Eliminating false data dependences using the Omega test. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation.* ACM Press, June 1992.

[31] D. Whalley R. Arnold, F. Mueller. Bounding worst-case instruction cache performance. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 172–181. IEEE Computer Society Press, December 1994.

[32] L. Sha, R. Rajkumar, and J. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Software Engineering*, 39:1175–1185, September 1990.

[33] K. Tindell. Using offset information to analyse static priority pre-emptively scheduled task sets. Technical Report YCS 182 (1992), Department of Computer Science, University of York, England, August 1992.

[34] K. Tindell, A. Burns, and A. Wellings. Allocating real-time tasks (an np-hard problem made easy). *The Journal of Real-Time Systems*, 4(2):145–165, June 1992.

[35] K. Tindell, A. Burns, and A. Wellings. An extendible approach for analysing fixed priority hard real-time tasks. *The Journal of Real-Time Systems*, 6(2):133–152, March 1994.

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

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

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

[39] M. Younis, T. Marlowe, and A. Stoyenko. Compiler transformations for speculative execution in a real-time system. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 109–117. IEEE Computer Society Press, December 1994.

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