To appear in 1994 IEEE Real-Time Systems Symposium.

Guaranteeing End-to-End Timing Constraints by Calibrating Intermediate Processes *

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

University of Maryland Technical Report
UMD CS-TR-3274, UMIACS-TR-94-58

May 1994

Abstract

This paper presents a comprehensive design methodology for guaranteeing end-to-end requirements of real-time systems. Applications are structured as a set of process components connected by asynchronous channels, in which the endpoints are the system's external inputs and outputs. Timing constraints are then postulated between these inputs and outputs; they express properties such as end-to-end propagation delay, temporal input-sampling correlation, and allowable separation times between updated output values.

The automated design method works as follows: First the end-to-end requirements are transformed into a set of intermediate rate constraints on the tasks, and new tasks are created to correlate related inputs. The intermediate constraints are then solved by an optimization algorithm, whose objective is to minimize CPU utilization. If the algorithm fails, a restructuring tool attempts to eliminate bottlenecks by transforming the application, which is then re-submitted into the assignment algorithm. The final result is a schedulable set of fully periodic tasks, which collaboratively maintain the end-to-end constraints.

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

1 Introduction

Most real-time systems possess only a small handful of *inherent* timing constraints which will "make or break" their correctness. These are called *end-to-end* constraints, and they are established on the systems' external inputs and outputs. Two examples are:

(1) *Temperature updates rely on pressure and temperature readings correlated within 10μs.*

(2) *Navigation coordinates are updated at a minimum rate of 40ms, and a maximum rate 80ms.*

But while such end-to-end timing parameters may indeed be few in number, maintaining *functionally correct* end-to-end values may involve a large set of interacting components. Thus, to ensure that the end-to-end constraints are satisfied, each of these components will, in turn, be subject to their own *intermediate* timing constraints. In this manner a small handful of end-to-end constraints may – in even a modest system – yield a great many intermediate constraints.

The task of imposing timing parameters on the functional components is a complex one, and it mandates some careful engineering. Consider example (2) above. In an avionics system, a "navigation update" may require such inputs as "current heading," airspeed, pitch, roll, etc; each sampled within varying degrees of accuracy. Moreover, these attributes are used by other subsystems, each of which imposes its own tolerance to delay, and possesses its own output rate. Further, the navigation unit may itself have other outputs, which may have to be delivered at rates faster than 800Hz, or perhaps slower than 400Hz. And to top it off, subsystems may share limited computer resources. A good engineer balances such factors, performs extensive trade-off analysis, simulations and sensitivity analysis, and proceeds to assigns the constraints.

These intermediate constraints are inevitably on the conservative side, and moreover, they are conveyed to the programmers in terms of constant values. Thus a scenario like the following is often played out: The design engineers mandate that functional units A, B and C execute with periods 65ms, 22ms and 27ms, respectively. The programmers code up the system, and find that C grossly over-utilizes its CPU; further, they discover that most of C's outputs are not being read by the other subsystems. And so, they go back to the engineers and "negotiate" for new periods – for example 60ms, 10ms and 32ms. This process may continue for many iterations, until the system finally gets fabricated.

This scenario is due to a simple fact: the end-to-end requirements allow many possibilities for the intermediate constraints, and engineers make what they consider to be a rational selection. However, the basis for this selection can only include rough notions of software structuring and scheduling policies – after all, many times the hardware is not even fabricated at this point!

Our Approach. In this paper we present an alternative strategy, which maintains the timing constraints in their end-to-end form for as long as possible. Our design method iteratively – and automatically – instantiates the intermediate constraints, all the while taking advantage of the leeway inherent in the end-to-end constraints. If the assignment algorithm fails to produce a full set of intermediate constraints, potential bottlenecks are identified. At this point an application analysis tool takes over, determines potential solutions to the bottleneck, and if possible, restructures the application to avoid it. The result is then re-submitted into the assignment algorithm.

Domain of Applicability. Due to the complexity of the general problem, in this paper we place the following restrictions on the applications that we handle.

Restriction 1: We assume our applications possess three classes of timing constraints which we call *freshness*, *correlation* and *separation*.

- A *freshness constraint* (sometimes called propagation delay) bounds the time it takes for data to flow though the system. For example, assume that an external output Y is a function of some system input X. Then a freshness relationship between X and Y might be: "If Y is delivered at time t, then the X-value used to compute Y is sampled no earlier than $t - 10ms$." We use the following notation to denote this constraint: "$F(Y|X) = 10$."
- A *correlation constraint* limits the maximum time-skew between several inputs used to produce an output. For example, if X_1 and X_2 are used to produce Y, then a correlation relationship may be "if Y is delivered at time t, then the X_1 and X_2 values used to compute Y are sampled no more than with $2ms$ of each other." We denote this constraint as "$C(Y|X_1, X_2) = 2$."
- A *separation constraint* constrains the jitter between consecutive values on a single output channel, say Y. For example, "Y is delivered at a minimum rate of 3ms, and a maximum rate of 13ms," denoted as $l(Y) = 3$ and $u(Y) = 13$, respectively.

While this constraint classification is not complete, it is sufficiently powerful to represent many timing properties one finds in a requirements document. (Our initial examples (1) and (2) are correlation and separation constraints, respectively.) Note that a single output Y_1 may – either directly or indirectly – be subject to several interdependent constraints. For example, Y_1 might require tightly correlated inputs, but may abide with relatively lax freshness constraints. However, perhaps Y_1 also requires data from an intermediate subsystem which is, in turn, shared with a very high-rate output Y_2.

Restriction 2: All subsystems execute on a single CPU. While a variant of our approach could be used in the multiprocessor case, it would greatly complicate the intermediate constraint-assignment problem.

Restriction 3: The entity-relationships within a subsystem are already specified. For example, if a high-rate video stream passes though a monolithic, compute-intensive filter process, this situation may easily cause a bottleneck. If our algorithm fails to find a proper intermediate timing constraint for the filter, the restructuring tool will attempt to optimize it as much as possible. In the end, however, it cannot redesign the system!

Finally, we stress that we are not offering a *completely automatic* solution. Even with a fully periodic task model, assigning periods to the intermediate components is a complex, nonlinear optimization problem which – at worst – can become combinatorially expensive. As for software restructuring, the specific tactics used to remove bottlenecks will often require user interaction.

Problem and Solution Strategy. We duly note the above restrictions, and tackle the intermediate constraint-assignment problem, as rendered by the following ingredients:

- A set of external inputs $\{X_1, \ldots, X_n\}$, outputs $\{Y_1, \ldots, Y_m\}$, and the end-to-end constraints between them.
- A set of intermediate component tasks $\{\tau_1, \ldots, \tau_l\}$, and their associated worst-case execution times $\{e_1, \ldots, e_l\}$.
- A task graph, denoting the communication paths from the inputs, through the tasks, and to outputs.

Solving the problem requires setting timing constraints for the intermediate components, so that all end-to-end constraints are met. Moreover, during any interval of time utilization may never exceed 100% (and in practice should be held as low as possible).

Our solution employs the following ingredients: (1) A periodic, preemptive tasking model (where it is the our algorithm's duty to assign the rates); (2) a buffered, asynchronous communication scheme, allowing us to keep down IPC times; (3) the period-assignment, optimization algorithm, which forms the heart of the approach; and (4) the software-restructuring tool, which takes over when period-assignment fails.

Related Work. This research was, in large part, inspired by the real-time transaction model proposed by Burns *et. al.* in [4]. While the model was formulated to express database applications, it can easily incorporate variants of our *freshness* and *correlation* constraints. In the analogue to freshness, a persistent object has "absolute consistency within t" when it corresponds to real-world samples taken within maximum drift of t. In the analogue to correlation, a set of data objects possesses "relative consistency within t" when all of the set's elements are sampled within a interval of time t.

We believe that in output-driven applications of the variety we address, separation constraints are also necessary. Without postulating a minimum rate requirement, the freshness and correlation constraints can be vacuously satisfied – by never outputting any values! Thus the separation constraints enforce the system's progress over time.

Burns *et. al.* also propose a method for deriving the intermediate constraints; as in the data model, this approach was our departure point. Here the high-level requirements are re-written as a set of constraints on task periods and deadlines, and the transformed constraints can hopefully be solved. There is a big drawback, however: the correlation and freshness constraints can inordinately tighten deadlines. E.g., if a task's inputs must be correlated within a very tight degree of accuracy – say, several nanoseconds – the task's deadline has to be tightened accordingly. Similar problems accrue for freshness constraints. The net result may be an over-constrained system, and a potentially unschedulable one.

Our approach is different. With respect to tightly correlated samples, we put the emphasis on simply getting the data into the system, and then passing through in due time. However, since this in turn causes many different samples flowing through the system at varying rates, we perform "traffic control" via a novel use of "virtual sequence numbering." This results in significantly looser periods, constrained mainly by the freshness and separation requirements. We also present a period assignment problem which is optimal – though quite expensive in the worst case.

This work was also influenced by Jeffay's "real-time producer/consumer model" [11], which possesses a task-graph structure similar to ours. In this model rates are chosen so that all messages "produced" are eventually "consumed." This semantics leads to a tight coupling between the execution of a consumer to that of its producers; thus it seems difficult to accommodate relative constraints such as those based on freshness.

Klein *et. al.* surveys the current engineering practice used in developing industrial real-time systems [12]. As is stressed, the intermediate constraints should be primarily a function of the end-to-end constraints, but should, if possible, take into account a sound real-time scheduling techniques. At this point, however, the "state-of-the-art" is the practice of trial and error, as guided by engineering experience. And this is exactly the problem we address in this paper.

The remainder of the paper is organized as follows. In Section 2 we introduce the application model and formally define our problem. In Section 3 we show our method of transforming the end-to-end constraints into intermediate constraints on the tasks. In Section 4 we describe the constraint-solver in detail, and push through a small example. In Section 5 we describe the application transformer, and in Section 6 we show how the executable application is finally built.

2 The Application Model

Our framework renders an application in an asynchronous task graph (ATG) format, where for a given graph $G(V, E)$:

- $V = P \cup D$, where $P = \{\tau_1, \ldots, \tau_n\}$, i.e., the the set of processes; and $D = \{d_1, \ldots, d_m\}$, a set of asynchronous, buffered channels. We note that the external outputs and inputs are simply typed nodes in D.
- $E \subseteq (P \times D) \cup (D \times P)$ is a set of directed edges, such that if $\tau_i \rightarrow d_j$ and $\tau_l \rightarrow d_j$ are both in E, then $\tau_i = \tau_l$. That is, each channel has a single-writer/multi-reader restriction.

The semantics of an ATG is as follows. Whenever a task τ_i executes, it reads data from all incoming channels d_j corresponding to the edges $d_j \rightarrow \tau_i$, and writes to all channels d_l corresponding to the edges $\tau_i \rightarrow d_l$. The actual ordering imposed on the reads and writes is inferred by the task τ_i's structure.

All reads and writes on channels are asynchronous and non-blocking. While a writer always inserts a value onto the end of the channel, a reader can (and many times will) read data from any location. For example, perhaps a writer runs at a period of 20ms, with two readers running at 120ms and 40ms, respectively. The first reader may use every sixth value (and neglect the others), whereas the second reader may use every other value.

While a channel is eventually implemented as circular, slotted buffer, a programmer treats it as unbounded, and accesses it using generic operations such as "**read**" and "**write**." After the constraint-assignment algorithm determines all of the processes' rates, a post-processing phase determines the actual buffer bounds, and then instantiate the "**read**" and "**write**" operations in each module.

This type of scheme allows us to minimize the overhead incurred when blocking communication is used, and concentrate exclusively on the assignment problem. In fact – as we show in the sequel – communication can be *completely unconditional*, in that we do not even require short locking for consistency. However, we pay a price for avoiding this overhead; namely, that the period assignments must ensure that no writer can overtake a reader currently accessing its slot.

Moreover, we note that our timing constraints define a system driven by time and *output requirements*. This is in contrast to reactive paradigms such ESTEREL [5], which are input-driven. Analogous to the "conceptual infinite buffering" assumptions, the rate assignment algorithm assumes that the external inputs are always fresh and available. The *derived* input-sampling rates then determine the *true* requirements on input-availability. And since an input X can be connected to another ATG's output Y, these requirements would be imposed on Y's timing constraints.

2.1 A Small Example

As a simple illustration, consider the system whose ATG is shown in Figure 1(A). The system is composed of six interacting tasks with three external inputs and two external outputs. The application's characteristics are as follows:

Freshness Constraints:	$F(Y_1\|X_1) = 30 \quad F(Y_1\|X_2) = 30$	$F(Y_2\|X_2) = 20 \quad F(Y_2\|X_3) = 15$
Correlation Constraints:	$C(Y_1\|X_1, X_2) = 3$	$C(Y_2\|X_2, X_3) = 4$
Separation Constraints:	$l(Y_1) = 18 \quad u(Y_1) = 31$	$l(Y_2) = 29 \quad u(Y_2) = 41$
Max Execution Times:	$e_1 = 6 \quad e_2 = 3 \quad e_3 = 3 \quad e_4 = 2 \quad e_5 = 3 \quad e_6 = 2$	

While the system is small, it serves to illustrate several facets of the problem: (1) There may be many possible choices of rates for each task; (2) correlation constraints may be tight compared to the allowable end-to-end delay; (3) data streams may be shared by several outputs (in this case that originating at X_2); and (4) outputs with the tightest separation constraints may incur the highest execution-time costs (in this case Y_1, which exclusively requires τ_1).

2.2 Solving the Problem

Guaranteeing the end-to-end constraints actually poses three sub-problems, which we define as follows.

Correctness: Let $\mathcal{C}$ be the set of derived, intermediate constraints and $\mathcal{E}$ be the set of end-to-end constraints. Then all system behaviors that satisfy $\mathcal{C}$ also satisfy $\mathcal{E}$.

Feasibility: The task executions inferred by $\mathcal{C}$ never demand an interval of time during which utilization exceeds 100%.

Schedulability: There is a scheduling algorithm which can efficiently maintain the intermediate constraints $\mathcal{C}$, and preserve feasibility.

In the problem we are addressing, the three issues cannot be decoupled. Correctness, for example, is often treated as verification problem using a logic such as RTL [10]. Certainly, given the ATG

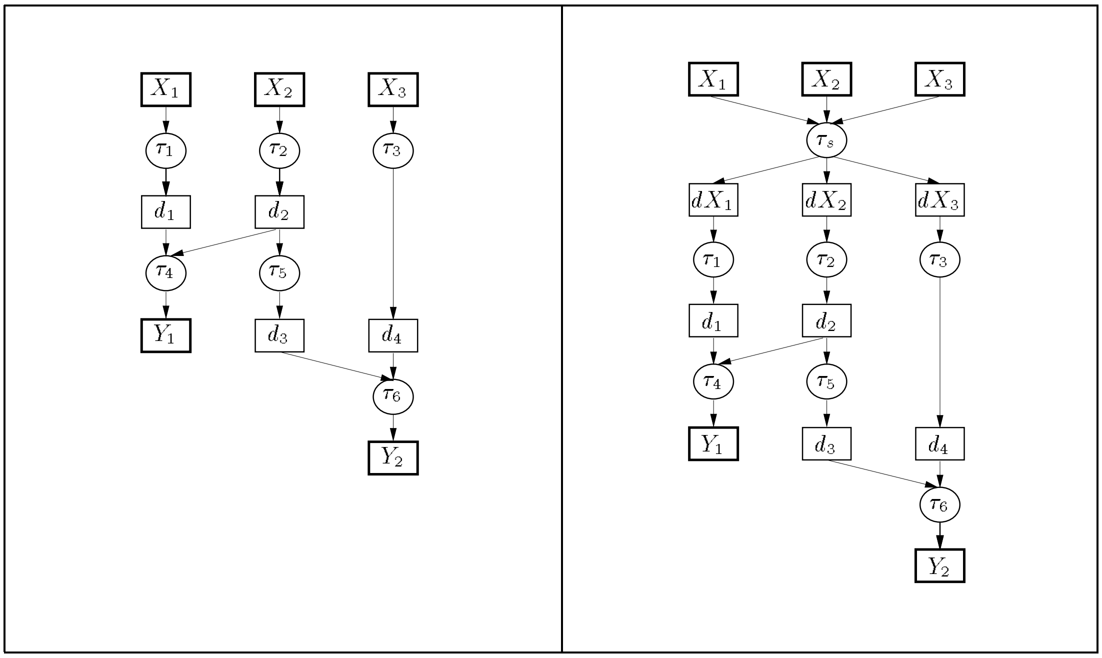


Figure 1: (A) A Task Graph (B) the Transformed Task Graph.

we could formulate $\mathcal{E}$ in RTL and query whether the constraint set is satisfiable. However, a "yes" answer would give us little insight into finding a good choice for $\mathcal{C}$ – which must, after all, be simple enough to schedule. Or, in the case of methods like model-checking ([2], etc.), we could determine whether $\mathcal{C} \Rightarrow \mathcal{E}$ is invariant with respect to the system. But again, this would be an *a posteriori* solution, and assume that we already possess $\mathcal{C}$. On the other hand, a system that is feasibile may still not be schedulable under a *known* algorithm; i.e., one that can be efficiently managed by a realistic kernel.

In this paper we put our emphasis on the first two issues. However, we have also imposed a task model for which the greatest number of efficient scheduling algorithms are known: simple, periodic dispatching with offsets and deadlines. Thus, we put structural limitations on the constraint set $\mathcal{C}$ so that the scheduling problem is reduced in complexity.

More formally, each task τ_i possesses a period T_i, an offset $\mathcal{O}_i \geq 0$ (denoting the earliest start-time from the start-of-period), and a deadline $D_i \leq T_i$ (denoting the latest finish relative to the start-of-period). Thus the interval $[\mathcal{O}_i, D_i]$ denotes the window of execution, with $W_i = D_i - \mathcal{O}_i$. The periods, deadlines and offsets make up constraint set $\mathcal{C}$.

The problem of scheduling a set of periodic real-time tasks on a single processor CPU has been studied for many years. Such a tasking model can be used to construct static calendar based schedules (e.g., [14]), or analyzed under a static preemptive priority scheme for schedulability. Our discussion for most part is independent of the underlying scheduling scheme. However, for the sake of simplicity we may assume an underlying static priority architecture. Static priority scheduling has been shown to be applicable to a number of variants of the periodic tasking model, such as

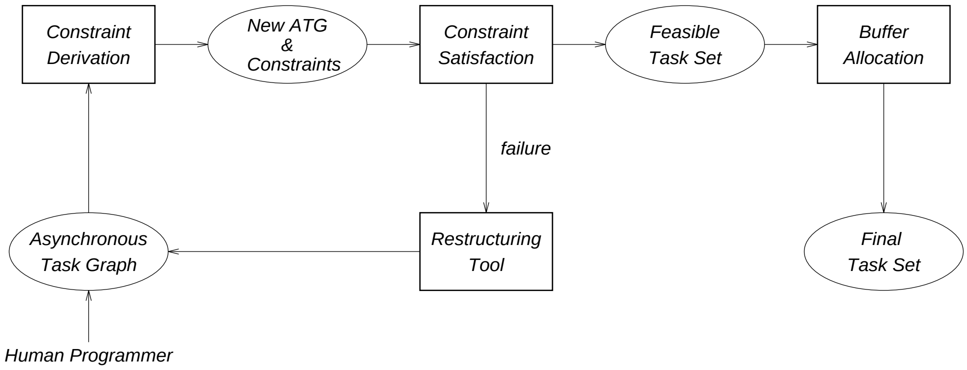


Figure 2: Overview of the Approach.

pre-period deadlines [3], precedence constrained sub-tasks [9], offsets [13] etc. A good overview to static priority scheduling may be found in [6].

Thus we focus our efforts on the correctness and feasibility problems. This is done in a four-step process, as shown in Figure 2: First the rate-based, intermediate constraints are derived, which may require creating new tasks to get tightly correlated inputs into the system. Next, a constraint-solver attempts to find a solution, by using the criterion of minimized CPU utilization. If a solution cannot be found, the restructuring tool alters the ATG to eliminate bottlenecks. Finally, the derived rates are used to reserve memory for the channels, and to instantiate the "**read**" and "**write**" operations.

3 Step 1: Deriving the Constraints

In this section we show the derivation process of intermediate constraints, and how they (conservatively) guarantee the end-to-end requirements. We start the process by synthesizing the intermediate correlation constraints, and then proceed to treat freshness and separation.

Synthesizing Correlation Constraints. Recall our example task graph in Figure 1(A), where the three inputs X_1, X_2 and X_3 are sampled by three separate tasks. If we wish to guarantee that τ_1's sampling of X_1 is correctly correlated to τ_2's sampling of X_2, we must pick short periods for both τ_1 and τ_2. Indeed, in many practical real-time systems, the correlation requirements may very well be tight, and way out of proportion with the freshness constraints. This typically results in periods that get tightened exclusively to accommodate correlation, which can easily lead to gross over-utilization. Engineers often call this problem "over-sampling," which is somewhat of a misnomer, since sampling rates may be tuned expressly for coordinating inputs. Instead, the problem arises from poor coupling of the sampling and computational activities.

Thus, our approach is to decouple these components as much as possible, and to create specialized samplers for related inputs. For a given ATG, the sampler derivation is performed by in the following manner.

foreach Correlation constraint $C_l(Y_l|X_{l_1},\ldots,X_{l_m})$

Create the set of all input-output pairs associated with C_l, i.e.,

$T_l := \{(X_{l_i}, Y_l)|X_{l_i} \in \{X_{l_1},\ldots,X_{l_m}\}\}$

foreach T_l, **foreach** T_k

If there's a common input X such that $\exists(X, Y_l) \in T_l\ \exists(X, Y_k) \in T_k$, and

if a path from X to Y_l shares a common task with the path from X to Y_k, then

Set $T_l := T_l \cup T_k$; $T_k := \emptyset$

foreach T_l, identify all associated sampling tasks, i.e.,

$S_l := \{\tau|(X, Y) \in T_l \wedge X \rightarrow \tau\}$

If $|S_l| \geq 1$, create a periodic sampler τ_{s_l} to take samples for inputs in T_l

Thus the incoming channels from inputs T_l to tasks in S_l are "intercepted" by the new sampler task τ_{s_l}.

Returning to our example Figure 1(A), since both correlated inputs share the center stream, the result is a single group of correlated inputs $\{(X_1, X_2, X_3)\}$. This, in turn, results in the formation of the single sampler τ_s. We assume τ_s has a low execution cost of 1. The new, transformed graph is shown at the right column of Figure 1(B).

As for the deadline-offset requirements, a sampler τ_{s_l} is constrained by the following trivial relationship

$$D_{s_l} - \mathcal{O}_{s_l} \leq t_{cor}$$

where t_{cor} is the minimum bound on all correlated inputs read by τ_{s_l}.

However, this by itself is not sufficient to ensure correlation requirements. Since correlated data can flow along different data streams, it may still reach the consumer task at different absolute times. (E.g., one stream may run faster than another.) Delay after all, is posited by the freshness requirement. For example, refer back to Figure 1, in which $F(Y_2|X_2) > F(Y_2|X_3)$. This disparity in freshness requirements causes a problem, since if task τ_6 attempts to satisfy the correlation constraints, it may have to violate the tighter freshness requirement. To ensure that the correlation requirements are also satisfied, we remove the "noise" that may exists between the different requirements. Thus, whenever a fresh output is required, we ensure that there are correlated data sets to produce it. This means that in our example, the freshness requirement $F(Y_2|X_2)$ is tightened to $F(Y_2|X_3)$.

Generally, whenever there is an output Y with freshness constraints $F(Y|X_1)$ and $F(Y|X_2)$, with X_1 and X_2 correlated as well, we set

$$F(Y|X_1), F(Y|X_2)\ := min\{F(Y|X_1), F(Y|X_2)\}$$

Synthesizing Freshness Constraints. Consider a freshness constraint $F(Y|X) = t_f$ between an input X and output Y. Recall that the freshness constraint requires that "for every output of Y at some time t, the value of X used to compute Y must have been read no earlier that time $t - t_f$." As the data flows through the task chain, a delay is incurred in processing the data

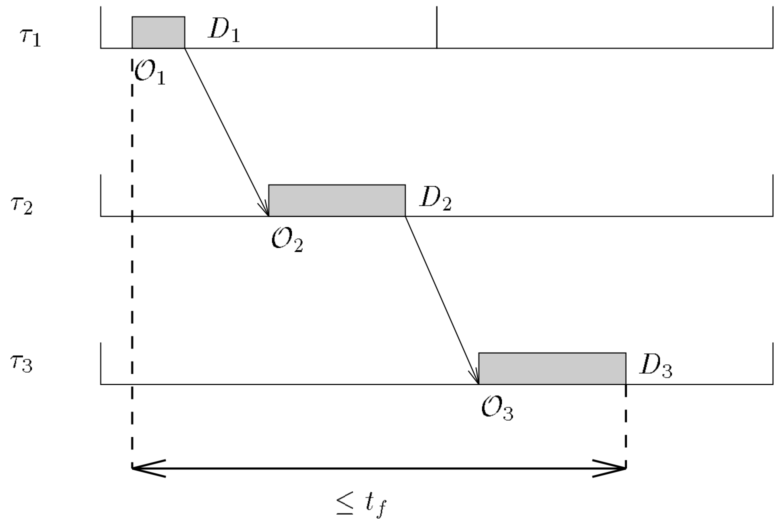


Figure 3: Freshness Constraints with Coupled Tasks

at each task in the chain, and in "handing over" of the data from one task to the other. The freshness requirement is satisfied if the cumulative delay does not exceed the freshness bound t_f. The delay due to processing in each task is dependent on the task's window of execution, whereas the delay suffered in hand over depends on the coupling between tasks. To optimize the waiting time associated with handover, we impose a harmonicity constraint between (producer,consumer) pairs; (i.e., pairs (τ_p, τ_c) where there are edges $\tau_p \rightarrow d$ and $d \rightarrow \tau_c$.)

Definition 3.1 (Harmonicity) *A task τ_2 is harmonic with respect to a task τ_1 if T_2 is exactly divisible by T_1 (represented as $T_2|T_1$*[1] *).*

To illustrate this refer to Figure 3 in which there are three tasks τ_1, τ_2 and τ_3 in a freshness chain. From the harmonicity assumption we have $T_3|T_2$ and $T_2|T_1$.

The harmonicity assumption allows us to achieve tight freshness requirements. Consider a freshness chain $\tau_1, \tau_2, \ldots, \tau_n$, where τ_n is the output task, and τ_1 is the input task. From the harmonicity assumption, we get $T_{i+1}|T_i$, for $1 \leq i < n$. Therefore, assuming that initially all tasks are invoked at time 0, for every invocation of the output task (τ_n) there is a simultaneous invocation of every task in the freshness chain. We derive the constraints based on the assumption that each task in the chain reads input data, processes it, and writes output data within this invocation. In that case, the worst case end-to-end delay is given by $D_n - \mathcal{O}_1$, and the freshness requirement if guaranteed if the following holds:

$$D_n - \mathcal{O}_1 \leq t_f$$

Note that we also require a precedence between each producer/consumer task pair. The simplest way of establishing precedence is by letting $D_i \leq \mathcal{O}_{i+1}$, for $1 \leq i < n$. However, the use of offsets

[1] $x|y$ iff $\exists \alpha ::\ \alpha y = x$ and $\alpha \geq 1$, where α is an integer.

and deadlines to enforce precedence like this has two drawbacks:

- A consumer task cannot start execution before its offset even if the producer task finishes its execution prior to its deadline.
- The end-to-end freshness bound t_f must be divided into a delay bound allowed at each task. It is not clear, how the slack should be distributed among intermediate tasks.

Instead, if the scheduler could enforce precedence, then we need not worry about the problem of distributing slack among intermediate tasks. The precedence is straightforward to handle in a calendar based scheduler. In a static priority scheduler, we need to ensure that (i) the producer has a higher priority, and (ii) the consumer is not made active before the producer. This is easily achieved by setting the offset of the consumer task equal to the offset of the producer task. Normally, this priority assumption is fine, since the deadline of the consumer task is higher than the producer, and should be assigned lower priority [3]. However, this is not desirable if the consumer task already has an offset requirement (e.g., due to separation constraints). In this case, the consumer task may have a smaller window of execution, and thus, a lower priority for it may not be the correct choice. As a result, we use the following rule of thumb:

> "If the consumer task is not an input or output task, i.e., if it does not have any other constraints on its offset, then its precedence requirement is deferred to the scheduler. Otherwise, the precedence requirement is satisfied through assignment of offsets."

Output Separation Constraints. Consider the separation constraints for an output Y, generated by some task τ_i. The window of execution $[\mathcal{O}_i, D_i]$ denotes the variability in the time an output can be made within a period. Thus, the separation constraints will be satisfied if the following holds true:

$$T_i + W_i \leq u(Y) \quad T_i - W_i \geq l(Y)$$

Example, Revisited. Revisiting the example, consider the constraints that arise from output separation requirements, which are induced on the output tasks τ_4 and τ_6. The derived constraints are presented below:

$$\begin{array}{ll} T_4 + (D_4 - \mathcal{O}_4) \leq u(Y_1) & T_4 - (D_4 - \mathcal{O}_4) \geq l(Y_1) \\ T_6 + (D_6 - \mathcal{O}_6) \leq u(Y_2) & T_6 - (D_6 - \mathcal{O}_6) \geq l(Y_2) \end{array}$$

In addition, we have the following constraints on each task.

$$\begin{array}{llll} \mathcal{O}_i + e_i \leq D_i, & \mathcal{O}_i \geq 0, & D_i \leq T_i, & 1 \leq i \leq 6 \\ \mathcal{O}_s + e_s \leq D_s, & \mathcal{O}_s \geq 0, & D_s \leq T_s, & \text{sampler task} \end{array}$$

Finally, consider the freshness constraints. There are four freshness constraints $F(Y_1|X_1) = 30$, $F(Y_1|X_2) = 30$, $F(Y_2|X_2) = 15$, and $F(Y_2|X_3) = 15$. Consider the freshness requirement $F(Y_1|X_1)$,

$F(Y_1\|X_1)$	$F(Y_1\|X_2)$	$F(Y_2\|X_2)$	$F(Y_2\|X_3)$
$D_4 - \mathcal{O}_s \leq 30$	$D_4 - \mathcal{O}_s \leq 30$	$D_6 - \mathcal{O}_s \leq 15$	$D_6 - \mathcal{O}_s \leq 15$
$D_s \leq D_1 - e_1$	$D_s \leq D_2 - e_2$	$D_s \leq D_2 - e_2,\ D_2 \leq D_5 - e_5$	$D_s \leq D_3 - e_3$
$D_1 \leq \mathcal{O}_4$	$D_2 \leq \mathcal{O}_4$	$D_2 \leq \mathcal{O}_6$	$D_3 \leq \mathcal{O}_6$
$T_4\|T_1, \quad T_1\|T_s$	$T_4\|T_2, \quad T_2\|T_1$	$T_6\|T_5, \quad T_5\|T_2, \quad T_2\|T_s$	$T_6\|T_3, \quad T_3\|T_s$
$\tau_s < \tau_1$	$\tau_s < \tau_2$	$\tau_s < \tau_2, \quad \tau_2 < \tau_5$	$\tau_s < \tau_3$

Table 1: Constraints due to Freshness Requirements

for which the data flows from through the task chain $\langle \tau_s, \tau_1, \tau_4 \rangle$. The end to end freshness requirement gives the constraint $D_4 - \mathcal{O}_s \leq 30$. Since τ_1 is an intermediate task we let the precedence $\tau_s < \tau_1$ be handled by the scheduler. However, we use the offset for τ_4 to handle the precedence $\tau_1 < \tau_4$. This leads to the constraints $D_1 \leq \mathcal{O}_4$ and $D_s \leq D_1 - e_1$. Similar inequalities are derived for the remaining freshness constraints, and the entire resulting set is shown in Table 1.

4 Step 2: Constraint Solver

The constraint solver forms the main component of our approach, and it generates a solution for the set of constraints derived from the high level timing requirements. The constraint solver must not only synthesize a satisfiable solution; it must also address the notion of feasibility as an optimization criteria. The feasibility aspect is addressed by minimizing the overall system utilization, as well as maximizing the window of execution for each task. Unfortunately, the non-linearity in the optimization criteria as well as the constraints leads to a complex non-linear optimization problem.

We present a solution to this problem which decomposes the problem into relatively tractable parts. The decomposition is motivated by the fact that the non-linear constraints in the problem, (i.e., the ones due to harmonicity) as well as the overall system utilization do not involve deadline and offset variables. This suggests the use of variable elimination to remove deadline and offset variables in deriving a transformed constraint set only involving period variables. The transformed constraint set may now be solved and optimized for minimum overall utilization. The solution for the periods is then used to derive the offsets and deadlines in a manner, which attempts to maximize the execution window of each task. An outline of our solution strategy is presented in Figure 4. The details of the three stages, i.e., variable elimination, deriving periods, and deriving offsets and deadlines are presented in the following subsections.

4.1 Elimination of Offset and Deadline Variables

We use an extension of Fourier-Motzkin variable elimination [7] to simplify our system of constraints. The Fourier Motzkin variable elimination is a linear programming technique which eliminates a variable from a set of linear constraints. Intuitively, it may be viewed as the projection of an n dimension convex object (described by the constraints) on to $n-1$ dimensions.

Algorithm 4.1 *Obtain offsets, deadlines, and periods for all tasks.*

/* C = Linear constraints on the task variables. */
/* H = Harmonicity constraints on the periods. */
/* $U = e_1/T_1 + e_2/T_2 + \ldots + e_n/T_n$. */

Step 1 Eliminate all variables other than T_i's from C to obtain a new set of constraints $\hat{C}$.

Step 2 Solve the set of constraints $\hat{C} \wedge H$ to minimize the objective function U.

Step 3 Let C' be the set of constraints obtained after substituting the values for T_i obtained in step 2. Solve C' for the offsets and deadlines, so as to maximize schedulability.

Figure 4: Top Level Algorithm to Obtain Task Characteristics.

Consider a system of constraints from which we want to eliminate variable x. Let $\alpha \leq x$, and $x \leq \beta$ be two constraints on x. Then, we can combine these two to infer $\alpha \leq \beta$, which is the projection of the two constraints. Therefore, if we combine every such pair of constraints on x, we obtain a new set of constraints in which x has been eliminated. The correctness of the method follows simply from the observation that if the new set of constraints has a solution, then there must be a value of x, which satisfies the original constraints.

The derived set of constraints have the property that any solution for task periods ensures that there is at least some solution for the offsets and deadlines which ensures correctness. The variable elimination method eliminates one variable at a time - generating a new set of constraints at each stage. As the constraints are generated, the algorithm tests for inconsistent constraints (e.g., $0 > 5$), and reports failure if any inconsistency is detected. In that case, the constraints are too tight to be satisfied, and application restructuring may be needed. In general, the algorithm can result in an exponential growth in the number of constraints generated. However, it is our belief that the use of an aggressive approach to remove redundant and trivially satisfiable constraints would help alleviate the problem significantly.

We illustrate the effect of variable elimination through the example application presented earlier. Due to space constraints, we do not trace the execution of the variable elimination algorithm; rather we present the final set of constraints that are generated. The derived constraints impose lower and upper bounds on task periods, and are shown in Table 2. We note that the final set of derived constraints have simple intuitive meaning, and other non-intuitive constraints generated due to variable interactions were found to be redundant.

In the final set of constraints, the constraints on the output tasks, i.e., τ_4 and τ_6 stem from the separation constraints, which impose both an upper and a lower bound on the period. The constraints on all other tasks are merely from the execution requirements. For example, within task τ_5's period we must be able to execute τ_s, τ_2 and τ_5 since they all are part of a freshness chain. Thus, $T_5 \geq e_s + e_2 + e_5 = 7$.

τ_s	τ_1	τ_2	τ_3	τ_4	τ_5	τ_6
$1 \leq T_s$	$7 \leq T_1$	$4 \leq T_2$	$4 \leq T_3$	$20 \leq T_4 \leq 29$	$7 \leq T_5$	$31 \leq T_6 \leq 39$

Table 2: Derived Constraints on Task Periods.

4.2 Deriving Task Periods

Once the deadlines and offsets have been eliminated from the constraints, we obtain a new set of constraints involving only the task periods. We now need to obtain a feasible period assignment which satisfies the derived constraints, as well as harmonicity constraints. Recall that our objective function is the overall system utilization which is given as $U = \sum \frac{e_i}{T_i}$.

In general, the derived constraints may be arbitrary linear constraints, and may require an expensive enumerative search algorithm to determine a feasible solution. In this paper, we restrict our solution to the special case in which the derived constraints only impose a lower/upper bound on a single task's period, as is the case in our example application. However, we believe that the algorithm is extensible to the more general case through the use of variable elimination.

Let l_i and u_i be the lower and upper bounds on a task period T_i. In addition, for each producer/consumer pair along a freshness chain, there is a harmonicity requirement. The harmonicity requirements may be succinctly represented as a directed acyclic graph, in which the nodes are the tasks and the edges represent harmonicity constraints. Each node also has a cost (its execution time) associated with it. An edge from τ_i to τ_j represents the constraint $T_j|T_i$. Let $Pred_i$ ($Succ_i$) denote the set of tasks which are predecessors (successors) of task τ_i, i.e., those tasks from which there is a directed path to (from) τ_i.

Clearly, this problem is a complex optimization problem in which non-linearity is imposed due to harmonicity and the optimization criteria. However, as will be evident, we also exploit this non-linearity to find an optimal solution. The key idea behind our optimization algorithm is pruning of the search space. The first step in the pruning process involves tightening the bounds on the periods. Subsequently, the task graph is simplified by merging nodes. By doing so, we restrict the number of variables in the optimization problem. These two steps are described below.

(1) Due to harmonicity, an edge $\tau_i \rightarrow \tau_j$ in the task graph implies that $T_i \leq T_j$. The first pruning takes place by propagating this information to tighten the period bounds. Thus, for each task τ_i, the bounds are tightened as follows:

$$l_i = \max\{l_k :: \ \tau_k \in Pred_i\} \qquad u_i = \min\{u_k :: \ \tau_k \in Succ_i\}$$

(2) The second step in the algorithm is to simplify the task graph. Consider a task τ_i, which has an outgoing edge $\tau_i \rightarrow \tau_j$. Suppose $u_i \geq u_j$, then the maximum value of T_i is constrained only by harmonicity restrictions. The task graph simplification is done by merging τ_i and τ_j, whenever it is safe to set $T_i = T_j$, i.e., the restricted solution space contains the optimal solution. The following two rules give the condition when it safe to perform this simplification.

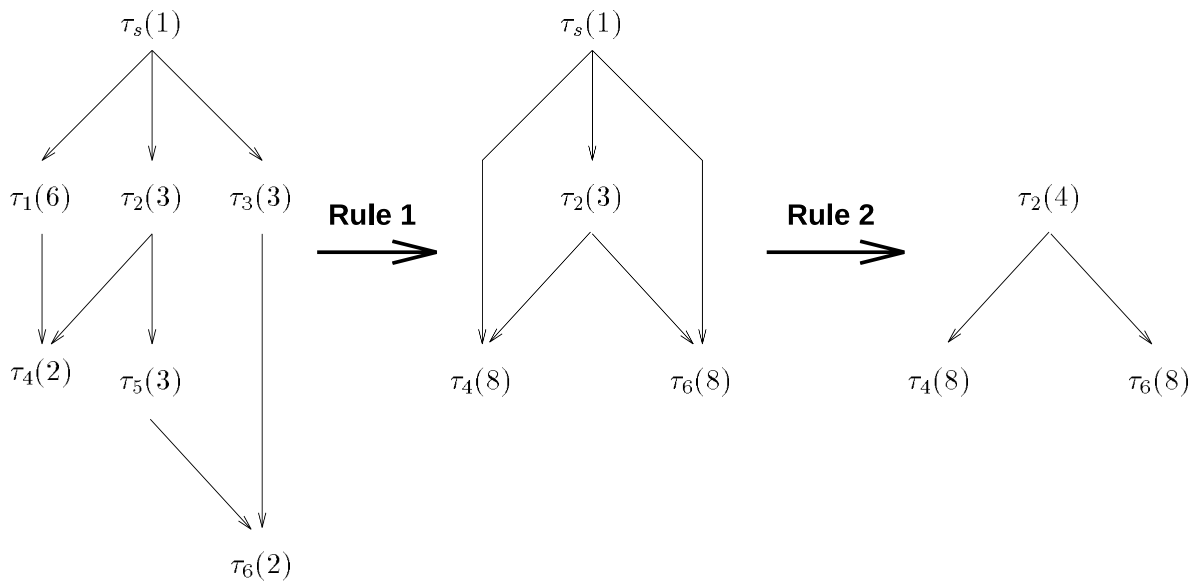


Figure 5: Task Graph for Harmonicity and its Simplification.

Rule 1: If a vertex τ_i has a single outgoing edge $\tau_i \rightarrow \tau_j$, then τ_i is merged with τ_j.

Rule 2: If $Succ_i \subseteq (Succ_j \cap \{\tau_j\})$ for some edge $\tau_i \rightarrow \tau_j$, then τ_i is merged with τ_j.

To illustrate these rules consider the task graph in Figure 5. The parenthesized numbers denote the costs of corresponding nodes. In the graph, the nodes τ_3, τ_5, and τ_1 have a single outgoing edge. Using **Rule 1**, we merge τ_3 and τ_5 with τ_6, and τ_1 with τ_4. In the simplified graph, $Succ_s = \{\tau_4, \tau_6, \tau_2\}$ and $Succ_2 = \{\tau_4, \tau_6\}$. Thus, we can invoke **Rule 2** to merge τ_s with τ_2.

The next stage in the pruning process is the use of harmonicity restrictions and utilization bounds to aggressively limit the search space. Let Φ denote the set of feasible solutions for T_i, and is initially given by $\Phi_i = \{T_i :: \; l_i \leq T_i \leq u_i\}$. The pruning takes place by successively refining and restricting Φ_i for each task. Algorithm 4.2 implements the pruning rules described below. The algorithm traverses the nodes in the task graph in a reverse topological ordering, and applies the pruning rules to restrict its own feasible set, as well as the feasible sets of its successors.

Pruning with Harmonicity Requirements: Consider any particular node τ_i in the task graph. Then, the feasible set of solutions for this node can be reduced by considering the harmonicity relationship with all its successor nodes.

$$\Phi_i := \{T_i \in \Phi_i :: \; (\forall \tau_k \in Succ_i)(\exists T_k \in \Phi_k :: T_k | T_i)\}$$

That is restrict Φ_i to those periods T_i for which there exists some period T_k in each of the successors such that $T_k | T_i$. The reduced feasible set of τ_i may now be propagated to all its successors τ_k by restricting their feasible space to those periods T_k which have some period $T_i \in \Phi_k$, such that $T_k | T_i$.

$$\Phi_k := \{T_k \in \Phi_k :: \ (\exists T_i \in \Phi_i :: \ T_k | T_i)\}$$

Pruning with Utilization Bounds: Let U_{max} be the upper bound on the utilization that we want to achieve. An any stage, a lower bound on the utilization for task τ_i is given by:

$$U_i^{min} = \frac{e_i}{T_i^{max}}, \text{where } T_i^{max} = \max\{T_i \in \Phi_i\}$$

Thus, if the lower bound on overall utilization $U_{min} (= \sum U_i^{min})$ exceeds U_{max} then there is no solution which satisfies the utilization bound. Now, consider a single task τ_ℓ, and consider a value $\hat{T}_\ell \in \Phi_\ell$. Define $\hat{T}_k$ for all other tasks as follows:

$$\hat{T}_k = \begin{cases} \max\{T_k \in \Phi_k :: \ T_k | \hat{T}_\ell\} & \tau_k \in Succ_l \\ \max\{T_k \in \Phi_k\} & \tau_k \notin Succ_l \end{cases}$$

Then, if $\hat{T}_\ell$ is the period for τ_ℓ, a lower bound on the utilization is given by:

$$U = \sum \frac{e_i}{\hat{T}_i}$$

Clearly, if $U > U_{max}$, then no feasible solution can be obtained with $\hat{T}_\ell$, and hence it may be removed from the feasible set.

Let us revert back to our example, and consider the reverse topological sort order τ_4, τ_6, τ_2, with execution time costs $8, 8$ and 4 respectively. The initial feasible sets are $\Phi_4 = \{T_4 :: \ 20 \leq T_4 \leq 29\}$, $\Phi_6 = \{T_6 :: \ 31 \leq T_6 \leq 39\}$, and $\Phi_2 = \{T_2 :: \ 4 \leq T_2 \leq 29\}$. Suppose the utilization bound $U_{max} = 1$.

Since τ_4 and τ_6 have no successors, and the utilization bounds are satisfied for all values, no restriction takes place. Now consider τ_2. The feasible set of values such that there is an integral multiple in each of Φ_4 and Φ_6 is:

$$\Phi_2 = \{4, 5, 6, 7, 8, 9, 11, 12, 13\}$$

Of these, after testing for utilization, we obtain the reduced set $\Phi_2 = \{9, 11, 12, 13\}$. This information is propagated to the successors to obtain $\Phi_4 = \{27, 22, 24, 26\}$ and $\{36, 33, 39\}$. The optimal solution in easily found to be $\tau_2 = 13, \tau_4 = 26, \tau_6 = 39$, giving a utilization of 0.82.

Unfortunately, the pruning algorithms do not totally avoid the search part, if we seek optimality. However, by carefully setting the utilization bound, we can limit the search time required, since the tighter the utilization bound, the greater is the pruning achieved. Thus, by starting with a low utilization bound, and successively increasing it if no solution is determined, we can limit the search time.

Algorithm 4.2 *Prune Feasible Search Space using Harmonicity and Utilization constraints.*

/* U_{max} = maximum allowable utilization. */
/* $T_i^{max} = \max\{T_i : T_i \in \Phi_i\}$ */
/* $U_{min} = \sum U_i^{min}$, where $U_i^{min} = e_i/T_i^{max}$ */

Sort the graph in reverse topological order. Let the sorted list be $L = \langle \tau_{i_1}, \tau_{i_2}, \ldots, \tau_{i_n} \rangle$.
for $j := 1$ to n **do** /* traverse the list */
    $\Phi_{i_j} = \{T_{i_j} \in \Phi_{i_j} :: \ (\forall \tau_k \in Succ_{i_j})(\exists T_k \in \Phi_k :: T_k | T_{i_j})\}$
    /* check utilization condition for each value in Φ_{i_j} */
    foreach $T_{i_j} \in \Phi_{i_j}$ **do**
        foreach $\tau_k \in Succ_{i_j}$ **do**
            $\hat{U}_k := \frac{e_k}{\hat{T}_k}$, where $\hat{T}_k = \max\{T_k \in \Phi_k :: \ T_k | T_{i_j}\}$
            $U_{min} := U_{min} - U_k^{min} + U_k'$
        end
        $U := U - U_{i_j}^{min} + e_i/T_{i_j}$
        if $U > U_{max}$ **then** $\Phi_{i_j} := \Phi_{i_j} - T_{i_j}$
    end
    /* Propagate restricted feasible set to all successors */
    foreach $\tau_k \in Succ_{i_j}$ **do**
        $\Phi_k := \{T_k \in \Phi_k :: \ (\exists T_{i_j} \in \Phi_{i_j} :: \ T_k | T_{i_j})\}$
    end
end

Figure 6: Pruning Feasible Space for Period Derivation.

4.3 Deriving Offsets and Deadlines

Once the task periods are determined, we need to revisit the constraints to find a solution to the deadlines and offsets of the periods. While the variable elimination method ensures that a feasible solution exists, we need to find a solution which maximizes schedulability. However, this is not easily achievable since no simple analytic solutions exist when tasks have both offsets and deadlines [13].

The variable elimination method allows us to select values for the variables in the reverse order in which they are eliminated. Suppose we eliminate variables $[x_1, x_2, \ldots, x_n]$ from a system of constraints in this order. Then, when variable x_i is eliminated, the remaining variables are $[x_{i+1}, \ldots, x_n]$. Also, note that the constraints at that stage can be written either as $\alpha \leq x_i$, or $x_i \leq \beta$, where α and β only contain variables $[x_{i+1}, \ldots, x_n]$. It is apparent that if $[x_{i+1}, \ldots, x_n]$ are already given values, then the constraints immediately give a lower and an upper bound on x_i, and any value of x_i within this range can be chosen.

We use this property of variable elimination in assigning offsets and deadlines to the tasks. As the variables are assigned values, each variable can be individually optimized. Recall that the feasibility of a task set requires that the task set never demands a utilization greater than one in

	τ_s	τ_1	τ_2	τ_3	τ_4	τ_5	τ_6
Period	13	26	13	39	26	39	39
Offset	0	0	0	0	21	0	13
Deadline	3	21	13	13	26	13	15
Execution Time	1	6	3	3	2	3	2

Table 3: Derived Task Set.

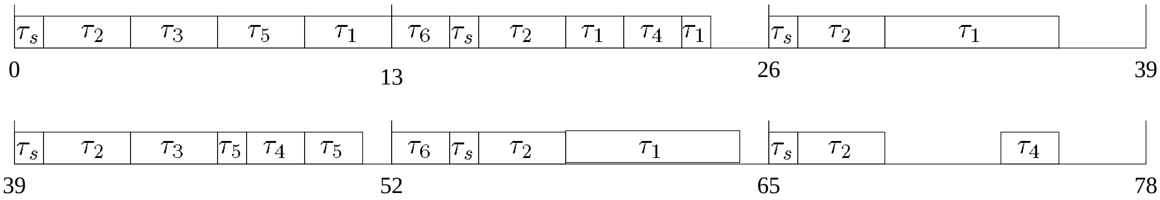


Figure 7: Feasible Schedule for Example Application

any time interval. We use a greedy heuristic, which attempts to maximize the window of execution for each task. For tasks which do not have an offset, this is straight forward, and can be achieved by maximizing the deadline. For input/output tasks which have offsets, we also need to fix the position of the window on the timeline. We do this by minimizing the offset for input tasks, and maximizing the deadline for offset tasks.

The order in which the variables are assigned is given by the following strategy: First, we assign the windows for each input task, followed by the windows for each output task. Then, we assign the offsets for each task followed by deadline for each output task. Finally, the deadlines for the remaining tasks are assigned in a reverse topological order of the task graph. Thus, an assignment ordering for the example application is given as $\{W_s, W_4, W_6, \mathcal{O}_s, D_4, D_6, D_5, D_3, D_1, D_2\}$. Notice that the variables must be eliminated in the reverse order of assignment. The final task set parameters, derived as a result of this approach, are shown in Figure 3. A feasible schedule for the task set is shown in Figure 7. We note that the feasible schedule can be generated using the fixed priority ordering $\tau_6, \tau_s, \tau_4, \tau_2, \tau_3, \tau_5, \tau_1$.

5 Step 3: Graph Transformation

When the constraint-solver fails, often replicating part of a task graph may prove useful in reducing the system's utilization. This benefit is realized by eliminating some of the tight harmonicity requirements, mainly by decoupling the tasks that possess common producers. As a result, the constraint derivation algorithm has more freedom in choosing looser periods for those tasks.

Recall the example application from Figure 1(B), and the constraints derived in Section 4. In the resulting system, the producer/consumer pair (τ_2, τ_5) has the largest period difference ($T_2 = 13$ and $T_5 = 39$). Note that the constraint solver mandated a tight period for τ_2, due to the coupled harmonicity requirements $T_4|T_2$ and $T_5|T_2$. Thus, we choose to replicate the chain including τ_2 from the sampler (τ_s) to data object d_2. This decouples the data flow to Y_1 from that to Y_2. Figure 8 shows the result of the replication.

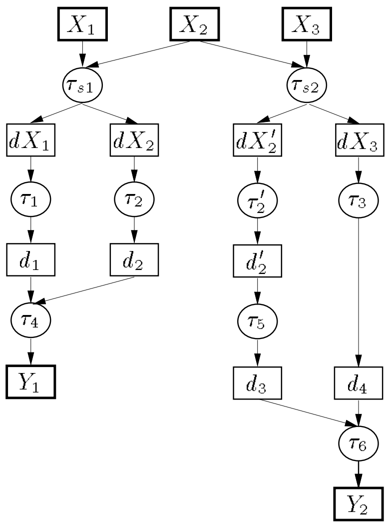


Figure 8: The Replicated Task Graph.

Running the constraint derivation algorithm again with the transformed graph in Figure 8, we obtain the following result.

	τ_{s1}	τ_1	τ_2	τ_4	τ_{s2}	τ_2'	τ_3	τ_5	τ_6
Periods	29	29	29	29	39	39	39	39	39
Execution Time	1	6	3	2	1	3	3	3	2

The transformed system has a utilization of 0.7215, which is significantly lower than that of the original task graph (0.8215).

The subgraph replication technique begins with selecting a producer/consumer pair which requires replication. There exist two criteria to select a pair depending on the goal. If the goal is reducing expected utilization, a producer/consumer pair with the maximum period difference is chosen first. On the other hand, if the goal is achieving feasibility, then we rely on the feedback from the constraint solver in determining the point of infeasibility.

After a producer/consumer pair is selected, the algorithm constructs a subgraph using a backward traversal of the task graph from the consumer. In order to avoid excessive replication, the traversal is terminated at the first confluence point. The resulting subgraph is then replicated and attached to the original graph.

The producer task in a replication may, in turn, be further specialized for the output it serves. For example, consider a task graph with two consumers τ_{c1} and τ_{c2} and a common producer τ_p. If we replicate the producer, we have two independent producer/consumer pairs, namely (τ_p, τ_{c1})

and (τ_p', τ_{c2}). Since τ_p' only serves τ_{c2}, we can eliminate all operations that only contribute to the output for τ_{c1}. This is done by a compiler technique called *dead code elimination* [1]. The same specialization is done for τ_p.

6 Step 4: Buffer Allocation

Buffer allocation is the final step of our approach, and hence applied to the feasible task graph whose timing characteristics are completely derived. During this step, the compiler tool determines the buffer space requirement for each data object and replaces reads and writes to a data object with system-provided macros to ensure the correct buffer management. Thus the goal of the buffer allocation is to enable the resultant system to combine a set of correlated data at any of its confluence points in a practical and realizable manner.

However, combining a set of correlated data at a given confluence point of a task graph is complicated, because (1) the producers and the consumer may all be running at different rates; and (2) the flow delays from a common sampler to the distinct producers may well be different. We approach these problems by allocating a multiple place buffer for each data object. As the data communication mechanism is one of the key components in our approach, we optimize it in terms of space and time requirements. To do so, we raise two issues.

(1) Finding a finite buffer size for each data object.

(2) Enabling a consumer task to take a set of correlated data by an efficient look-up in the multiple place buffers.

Our solution to these issues relies on the following facts.

Fact 6.1 *Let (τ_p, τ_c) be a producer/consumer pair, and d be a data object between them. When τ_c is about to read from d, any data items τ_p has put into d since the beginning of the τ_c's current period, are valid with respect to the freshness constraint imposed on the flow.*

Figure 3 pictorially illustrates this situation. By the induction on the chain of producer/consumer pairs, it can be easily proved that this is the case.

Fact 6.2 *For $1 \leq i \leq n$, let (τ_{p_i}, τ_c) be the i^{th} producer/consumer pair, and d_i be a data object of the pair. Suppose all d_is must be correlated. Then a data set consisting only of the data items, each of which τ_{p_i} put into d_i first within the τ_c's current period, is tightly correlated. (They were read in the same period by the sampler.)*

This is trivially the case, because all data items in a correlated set are read by a common sampler and any pair of producer/consumer has harmonic periods.

Based on these facts, the solution strategy is:

> "Whenever a consumer reads from a channel, it uses the *first* item that was generated within its current period."

As a result, when a producer/consumer pair (τ_p, τ_c) and their shared data object d are given, the communication mechanism is realized by the following techniques:

(1) The data object d is implemented with s number of buffers where $s = T_c/T_p$.

(2) The producer τ_p circularly writes into d: It begins with slot 0, and then writes into the next until it fills slot $s-1$; then it goes back to slot 0.

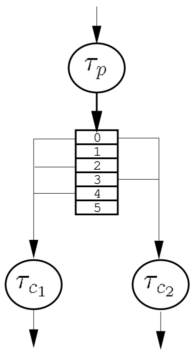
(3) The consumer τ_c reads only from slot 0.

However, the technique is incomplete for the situation where a data object is read by multiple consumers. For $1 \le i \le n$, let (τ_p, τ_{c_i}) be the i^{th} producer/consumer pair, and d be a data object shared by the consumers τ_{c_i}. Also, let $L = LCM_{1 \le i \le n}(T_{c_i})$.[2] The general techniques are:

(1) The data object d is implemented with s number of buffers where $s = L/T_p$.

(2) The producer τ_p circularly writes into d in exactly the same way as above.

(3) The consumer τ_{c_i} reads circularly from slots $(0, T_{c_i}, \ldots, m \cdot T_{c_i})$ where $m = L/T_{c_i} - 1$.



$$\begin{aligned} T_p &= 10 \\ T_{c_1} &= 20 \\ T_{c_2} &= 30 \end{aligned}$$

Figure 9: A Task Graph with Buffers.

Figure 9 shows a task graph after the buffer allocation. In the graph, there are two consumer tasks τ_{c_1}, τ_{c_2} running with periods 20 and 30, respectively and one producer running with period 10. Thus, the data object requires a 6 ($6 = LCM(20, 30)/10$) place buffer, and τ_{c_1} reads from slots (0, 2, 4) while τ_{c_2} reads from slots (0, 3).

After the buffer allocation, the compiler tool expands each data object into a multiple place buffer, and replaces each read and write operations with macros that perform proper pointer updates.

7 Conclusion

We have presented a four-step design methodology to help synthesize end-to-end requirements into full-blown real-time systems. Our framework can used as long as the following ingredients are

[2] LCM is the least common multiple.

provided: (1) the entity-relationships, as specified by an asynchronous task graph abstraction; and (2) end-to-end constraints imposed on freshness, input correlation and allowable output separation. This model is sufficiently expressive to capture the temporal requirements – as well as the modular structure – of many interesting systems from the domains of avionics, robotics, and control and multimedia computing.

However, the asynchronous, fully periodic model does have its limitations; for example, we cannot support high-level blocking primitives such as RPCs. On the other hand this deficit yields significant gains; e.g., handling streamed, tightly correlated data solely via the "virtual sequence numbers" afforded by the rate-assignments.

There is much work to be carried out. First, the constraint derivation algorithm can be extended to take full advantage of a wider spectrum of timing constraints, such as those encountered in input-driven, reactive systems. Also, we can harness finer-grained compiler transformations such as *program slicing* to help transform tasks into read-compute-write-compute phases, which will even further enhance schedulability. We have used this approach in a real-time compiler tool [8], and there is reason to believe that its use would be even more effective here.

Finally, perhaps the greatest challenge lies in incorporating scheduling decisions into the constraint solver. We believe such policy-specific strategies can be used to significantly help in pruning the search space.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison Wesley Publishing Company, 1986.

[2] R. Alur, C. Courcoubetis, and D. Dill. Model-Checking for Real-Time Systems. In *Proc. of IEEE Symposium on Logic in Computer Science*, 1990.

[3] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. Hard Real-Time Scheduling: The Deadline Monotonic Approach. In *Proceedings Eighth IEEE Workshop on Real-Time Operating Systems and Software*, pages 133–137, May 1991.

[4] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. Data consistency in hard real-time systems. Technical Report YCS 203 (1993), Department of Computer Science, University of York, England, June 1993.

[5] G. Berry, Sabie Moisan, and Jean-Paul Rigault. ESTEREL: Towards a synchronous and semantically sound high level language for real time applications. In *Proceedings IEEE Real-Time Systems Symposium*, pages 30–37. IEEE Computer Society Press, December 1983.

[6] Alan Burns. Preemptive priority based scheduling: An appropriate engineering approach. In Sang Son, editor, *Principles of Real-Time Systems.* Prentice Hall, 1994.

[7] G. B. Dantzig and B. C. Eaves. Fourier-Motzkin Elimination and its Dual. *Journal of Combinatorial Theory (A)*, 14:288–297, 1973.

[8] 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.

[9] M. G. Harbour, M. H. Klein, and J. P. Lehoczky. Fixed Priority Scheduling of Periodic Tasks with Varying Execution Priority. In *Proceedings, IEEE Real-Time Systems Symposium*, pages 116–128, December 1991.

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

[11] Kevin Jeffay. The real-time producer/consumer paradigm: A paradigm for the construction of efficient, predictable real-time systems. In *ACM/SIGAPP Symposium on Applied Computing*, pages 796–804. ACM Press, February 1983.

[12] M. Klein, J. Lehoczky, and R. Rajkumar. Rate-monotonic analysis for real-time industrial computing. *IEEE Computer*, pages 24–33, January 1994.

[13] K. W. 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.

[14] Jia Xu and David Parnas. Scheduling processes with release times, deadlines, precedence and exclusion relations. *IEEE Transactions on Software Engineering*, 16(3):360–369, March 1990.