Designing Distributed Real-Time Systems with Decomposition of End-to-End Timing Constraints

양극단 지연시간의 분할을 이용한 분산 실시간 시스템의 설계

홍 성 수
(Seong Soo Hong)

Abstract : In this paper, we present a resource conscious approach to designing distributed real-time systems as an extension of our original approach [8][9] which was limited to single processor systems. Starting from a given task graph and a set of end-to-end constraints, we automatically generate task attributes (e.g., periods and deadlines) such that (i) the task set is schedulable, and (ii) the end-to-end timing constraints are satisfied. The method works by first transforming the end-to-end timing constraints into a set of intermediate constraints on task attributes, and then solving the intermediate constraints. The complexity of constraint solving is tackled by reducing the problem into relatively tractable parts, and then solving each sub-problem using heuristics to enhance schedulability. In this paper, we build on our single processor solution and show how it can be extended for distributed systems. The extension to distributed systems reveals many interesting sub-problems, solutions to which are presented in this paper. The main challenges arise from end-to-end propagation delay constraints, and therefore this paper focuses on our solutions for such constraints. We begin with extending our communication scheme to provide tight delay bounds across a network, while hiding the low-level details of network communication. We also develop an algorithm to decompose end-to-end bounds into local bounds on each processor of making extensive use of relative load on each processor. This results in significant decoupling of constraints on each processor, without losing its capability to find a schedulable solution. Finally, we show, how each of these parts fit into our overall methodology, using our previous results for single processor systems.

Keywords : design methodology, fiming constraints, optimization, distributed, real-time systems

I. Introduction

Recent maturity of real-time scheduling and analysis techniques (e.g., rate-monotonic and other variants of static priority scheduling), and their incorporation into industrial systems have established a periodic task model as an essential vehicle for real-time systems development. In such a periodic task model, tasks are made to run repeatedly at fixed rates interacting with each other in a controlled manner. However, as real-time applications become more diversified and complex, modeling them as a set of independent, periodic tasks gets more difficult. Many of today's real-time systems such as multimedia, manufacturing and vehicle control systems require sharing resources and data, synchronizing task executions, and timely flow of data through multiple data paths on a distributed platform. These systems often possess constraints which are established between external inputs and outputs such as *end-to-end* propagation delays, input and output jitter, as well as rate constraints.

In this paper we address the problem of transforming a high-level real-time system design into a set of (schedulable) periodic tasks. The current work extends our original solution [8][9], which was limited to single processor environments. The solution is derived by first deriving a set of intermediate constraints on task attributes (e.g., periods and deadlines) from the original end-to-end requirements, and then using a constraint solver to solve the constraints, such that the final derived task set is schedulable. As can be expected, the extension to a distributed system environment reveals many interesting and difficult challenges. First, incorporating schedulability criteria in the constraint solver becomes more difficult, since we must now ensure schedulability on each host as well as the network. Second, we must effectively decouple the constraints for different processors and the network to avoid being forced to solve a very complex global optimization problem. Third, the low-level networking details for communication across the network should be hidden from the programmer; yet the overheads due to network interrupts, and problems due to limited buffer sizes of network adaptors must be suitably accounted.

In extending our approach to distributed systems, we find that the main difficulties arise from end-to-end freshness constraints. For a single processor system, we could defer precedence constraints to the scheduler, thereby simplifying the problem; we now have to explicitly enforce precedence through task phasing, resulting in additional variables and complexity for the constraint solver. In addition, we have a new

접수일자 : 1996. 10. 2., 수정완료 : 1997. 10. 27.
홍성수 : 서울대학교 공과대학 전기공학부

sub-problem of decomposing end-to-end freshness bound into a local bound for each delay component along the data path. We present a solution to this new sub-problem, and show how it can be integrated into the overall approach. We also show how network communication can be synthesized using a specific network controller for Controller Area Network (CAN).

The extended solution inherits all the benefits of the original approach: (1) It provides programmers with a rapid prototyping tool which helps them build a running prototype fast and locate bottlenecks in it; and (2) it helps programmers fix and optimize the faulty design for both correctness and performance. This is possible not only because the system traceability is maintained through the use of a semi-automatic approach, but also because the constraint solver itself generates various performance metrics.

In generating the solution, we do not address the problem of scheduling the derived periodic tasks; instead we rely on existing scheduling methods to do the job for us. Likewise, we assume that the tasks and their communications are already specified using a task graph model. We also assume that the allocation of tasks to processing hosts is given, that is we do not address the task allocation problem. Finally, we perform all our timing analysis on a notional global clock, and assume that the local clock on each host is synchronized with respect to this global time.

1. Related work

Real-time system design and scheduling have been fertile areas of research in the last decade. We refer the readers to [10] for an overview of design methods and [3][17] for an overview of real-time scheduling. There has been relatively less effort in the integration of design and scheduling, and specifically the derivation of task periods and deadlines from end-to-end constraints. In [1] and [19] similar problems are addressed, but the focus was more on schedulability analysis, and less on the derivation of task parameters. There have also been some studies on the decomposition of end-to-end deadlines into local task deadlines [11][2][7][18]. In [11][2], various strategies for priority assignment are proposed, all in essence using a proportionate deadline assignment, and then using deadline monotonic priorities. In [7] a heuristic algorithm is presented similar in spirit to our deadline decomposition algorithm, but does not attempt to isolate bottleneck tasks or provide feedback on failure. Finally, [16] also addresses deadline decomposition, but does not consider resource contention. Their performance metric involves maximizing the minimum (normalized) laxity for each task.

A similar end-to-end scheduling problem is encountered in real-time communication over multiple hops, where an end-to-end delay bound must be decomposed into deadlines at each hop of the network [6][12]. The end-to-end deadline scheduling problem is very similar to decomposition of freshness bounds. However, we could not use the solutions available in the literature since unlike the problems studied above, we had multiple interacting end-to-end deadlines, instead of a single one. Furthermore, we believe that our objective function gives much better answers, as it incorporates knowledge about relative load on each processor, and the amount of utilization available to each task.

2. Remainder of the paper

The remainder of the paper is organized as follows. Section 2 presents the application and system models and defines our problem. Section 3 briefly reviews the single-processor approach our distributed extension is based on, and gives an overview of the extended solution. Section 4 shows how the intermediate constraint system is derived from a given application and end-to-end timing constraints. Section 5, which is the crux of our approach, discusses the constraint solver algorithm and demonstrates its strategy with a simple but representative example. Section 6 details the implementation of a network channel. Finally, in Section 7 we conclude the paper with future research directions.

II. Problem description and solution overview

In this section, we elaborate on the system and application model.

1. System and network model

We consider a distributed system of processing hosts, connected together by a suitable communication network. The sensors and actuators used for external inputs and outputs are considered directly attached to hosts via the host's local IO bus. For simplicity we assume that the local clocks are all synchronized to a global time-base, and all our analysis is done with respect to this global time-base. This assumption does not impose a serious restriction, since we can tighten up the end-to-end constraints to accommodate local clock drift.

The communication network should be capable of guaranteeing bounded message transfer delays [23]. In our application model, messages are generated periodically, and many papers have addressed the problem of guaranteeing real-time deadlines for periodically generated messages, on a variety of networks and protocols, for example FDDI [4], slotted-ring [15], DQDB [18] and CAN [21]. For the purpose of analysis, we assume that a scheduling mechanism exists that can verify whether a given set of periodic message streams, with deadlines less than period, are schedulable. The deadline reflects the time duration starting from the time the message is queued for transmission at the sender and finishing when the message is delivered on the receiver's buffers [22][21].

2. Application model: partitioned asynchronous task graph

An application is rendered in our framework as an asynchronous task graph (ATG), as in [9], but extended to incorporate communication across a network. For a given graph $G(V,E)$:

- $P \cup D \cup C$, where $P=\{\tau_1, \ldots, \tau_n\}$ i.e., the set of *periodic* tasks, $D=\{d_1, \ldots, d_p\}$ a set of asynchronous, buffered *data* channels, and $C=\{c_1, \ldots, c_m\}$ a set of

network channels. The external outputs and inputs are simply typed data channels in D.

- $E \subseteq (P \times D) \cup (D \times P) \cup (P \times C) \cup (C \times P)$ is a set of directed edges between tasks and channels (data or network). $\tau_i \rightarrow d_j$ denotes task τ_i's write access to data channel d_j and $d_j \rightarrow \tau_i$ does τ_i's read access to d_j. We do not permit multiple incoming edges to a single channel in order to avoid a consistency problem[1]. A channel with multiple incoming edges can be translated into multiple, replicated ones, each of which is connected to a single incoming edge.
- Each (computation) task $\tau_i \in P$ has the following attributes: a period T_i, an offset $O_i \geq 0$ (denoting the earliest start-time from the start-of-period), a deadline $D_i \leq T_i$ (denoting the latest finish-time relative to the start-of-period), a maximum execution time e_i, and a phase $\phi_i \geq 0$ (denoting the invocation time for the first instance of the periodic task). The interval $[O_i, D_i]$ constrains the window W_i of execution for the task, where $W_i = D_i - O_i$. Intuitively, the phase of a periodic task is the time when its first request is made since the entire system was made to run. Note that all periodic tasks need not be initiated at the same time, unlike real-time systems found in [14][13].
- Network channels abstract the underlying network communication, and carry periodic message streams from a sender task to one or more receiver tasks. Each message stream m_i on channel c_i is called a *network task*, and has the following attributes: a period T_i^m, a deadline $D_i^m \leq T_i^m$, maximum message transmission time e_i^m, and a phase ϕ_i^m. Simply, a network channel is associated with a network task. Without loss of generality, the offset of all network tasks is 0 . Note that a message can be transmitted at any time within its period, so long as it is made available by the sender.

Structurally, an ATG may be viewed as a set of disjoint sub-graphs, if we remove the vertices corresponding to network channels (and all edges incident on those vertices). We call each such disjoint sub-graph a *partition*. Each partition is mapped to a single host. The mapping of partitions to hosts defines the allocation of tasks, and we assume that this mapping is already defined. Thus the task graph represents static task allocation, and its structure is not changed at run-time. The network channels delineate the boundary between tasks on two partitions, and denote the message transfer over the network.

The semantics of an ATG is as follows. Whenever a task τ_i executes, it reads data from all incoming data and/or network channels (i.e., channels from which there is directed edge to τ_i), and writes to all outgoing data and/or network channels (i.e., channels to which there is a directed edge from τ_i). The actual ordering imposed on the reads and writes is inferred by the task τ_i's structure.

To programmers, all reads and writes on data and network channels are asynchronous and non-blocking. These channels can even be considered as unbound buffers, and they are instantiated with finite blocks of memory later in a transparent manner. Network channels abstract the communication between two hosts, and they are implemented as sets of circular, slotted buffers, one at the sender's host, and one at each receiver's host. Data is transferred from the sender's buffers to the network by a virtual communication task. On the receiver's side the data retrieved from the network is directly placed into the appropriate location in the receiver's buffers.

Both of the local and network communication schemes present the same non-blocking communication model to designers by successfully hiding implementation details of the communication scheme. Furthermore, it also allows designers to avoid the complex analysis needed for blocking communications, and to focus exclusively on the assignment problem. As we showed in our previous work in [9], it is the period assignments that allows for asynchronous communication by ensuring that no writer can overtake a reader currently accessing its slot. On the other hand, unlike in [9], complete asynchrony can not be achieved since network channels require short locking between sender tasks and interrupt handlers. However, such details are still hidden from designers.

3. End-to-End timing constraints

The end-to-end real-time requirements postulate four different types of constraints as follows: (i) A *freshness constraint* $F(Y|X) = t_f$ reflects the constraint "If an output Y is delivered at time t , then the input X used to compute Y is sampled no earlier than $t - t_f$," (ii) A *correlation constraint* $C(Y|X_1, X_2) = t_{cor}$ says that "if Y is delivered at time t, then the X_1 and X_2 values used to compute Y must be sampled within t_{cor} time of each other," (iii) *Output jitter constraints* $J_Y^l = t_1$ and $J_Y^u = t_2$ reflect the requirement that two successive outputs of Y must be separated by no less than t_1 and no more than t_2 time units, and (iv) the *output rate constraints* $T_Y^l = t_1$ and $T_Y^u = t_2$ reflect the requirement that output Y must be delivered at a minimum average rate of once every t_2 time units, and a maximum average rate of once every t_1 time units.

4. A simple example

We present a simple example task graph (shown in Figure 1). This example ATG will be used throughout the paper to illustrate the solution process. The example ATG consists of two external inputs X_1, and X_2 and two outputs Y_1 and Y_2. There are four tasks in the system - which collaboratively use the external inputs to produce external outputs. Each task forms a separate

1) Two interleaved write accesses to a shared channel may cause it to be in an inconsistent state, since we cannot predict which write access will be final.

partition, and we assume that τ_1 and τ_2 are mapped to one host, and τ_3 and τ_4 are mapped to the other host.

We impose a minimal set of constraints on the ATG comprising of minimum rate for the outputs, and end-to-end freshness bounds on the different input-output pairs. We find that this minimal constraint set is sufficient to illustrate the key solution techniques presented in the paper, as well as being simple enough to guide the reader through the solution process. However, we caution the reader that the absence of output jitter and correlation constraints, as well as more intricate communication patterns, greatly simplify some stages of the solution process - specifically the assignment of deadlines, offsets, and phases.

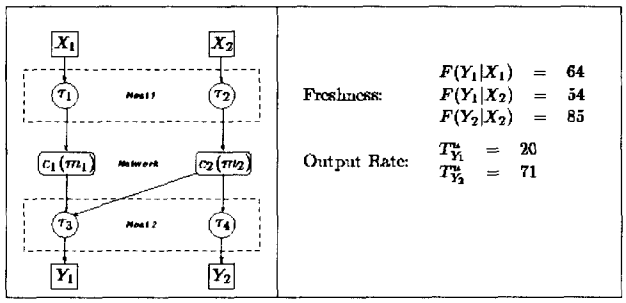

Fig. 1. Example ATG structure and the End-to-End constraints.

III. Review of the single-processor approach

The distributed extension we present in the sequel is based on the single-processor approach, hence its brief review is relevant. In short, the approach attempts to derive task-specific attributes in a semi-automatic manner. It accepts an application's task graph and end-to-end timing requirements on the application's inputs and outputs. It then translates them into a set of nonlinear inequalities which possess task-specific attributes as free variables. It proceeds to solve these inequalities, while minimizing the application's CPU demand. We first define the task-specific attributes and explain the constraint solving process.

1. Task-specific attributes

Each task in a task graph is a periodic one which is repetitively invoked and executed at a fixed rate. The periodic task has three kinds of timing attributes, namely *period, offset* and *deadline* which are denoted by T_i, O_i and D_i, respectively.

There was an important assumption made on task periods, in [8]. Suppose task τ_i writes into a channel and task τ_j reads from it. Then we call τ_i a *producer* task and τ_j a *consumer* task. For a given pair of producer/consumer tasks (τ_i, τ_j), the approach required that period T_j be an integral multiple of period T_i. We called this *harmonicity requirement* and denoted it by "$T_j | T_i$". The harmonicity requirement allowed for significant benefits in task communication and task scheduling, as shown in [8][9].

2. Steps of the approach

For a given task graph possessing n tasks $\{\tau_1, \tau_2, \ldots, \tau_n\}$ and the end-to-end requirements oninputs and outputs, the approach derived T_i, O_i and D_i in two steps, as below.

Step 1 : It derived a set of nonlinear constraints on task-specific attributes T_i, O_i and D_i such that the derived constraints implied the original end-to-end constraints.

Step 2 : It solved the derived nonlinear constraints such that chances of obtaining a schedulable task set were maximized. We used the following objective functions.

Step 2 involved solving a nonlinear optimization problem which is conjectured to be an NP-hard problem. The primary source of complexity for the problem was the non-linearity introduced by the harmonicity requirements on period variables. Therefore, in order to tackle this complexity, we adopted the following approach:

(1) First the entire constraint set was reduced to constraints involving period values only, using Fourier variable elimination (see appendix) method for linear constraints.

(2) The constraints on periods were then solved and optimized for minimum utilization using a combination of heuristics derived from harmonicity requirements.

(3) Finally, with the values of T_i's plugged into the original constraint set, a solution for offsets and deadlines was obtained using a greedy heuristic of maximizing the window of execution of tasks.

The primary benefit of the approach was the decomposition of a complex problem into relatively tractable sub-problems. In the case above, we got two simplified sub-problems: one involving period variables, and one involving deadline and offset variables. Furthermore, each sub-problem had a simpler and better defined objective function: minimizing utilization for period assignment, and maximizing execution windows for deadline and offset assignment.

3. Solution overview of the extended approach

As in [9], our objective is to derive a set of intermediate constraints C, which when satisfied ensure that the set of end-to-end constraints ε are satisfied as well. The constraint set C is merely a set of constraints on the task (including network tasks) attributes (the periods, deadlines, offsets, and phases). The final result is a set of periodic tasks, and a set of periodic message streams, which if found to be schedulable, guarantee that the end-to-end constraints will be satisfied. The problem of scheduling of periodic real-time tasks and periodic real-time message streams is well studied in the literature, and is not addressed in this paper. Instead, we concentrate on the derivation of task attributes.

Our solution process is carried out in several steps as in our single processor solution. We begin with the derivation of intermediate constraints from the original end-to-end requirements. Following that, the constraint-solver is invoked to solve the intermediate constraints,

using heuristics to achieve schedulability of the derived task set. Once the constraint solver returns a feasible solution, the channels are implemented by instantiating the **"read"** and **"write"** operations, creating interrupt handlers for network communication, and estimating the overheads due to interrupts in the schedulability analysis. The final result must then be subjected to schedulability analysis, which we do not address.

IV. Deriving the Intermediate constraints

The first step in our approach is to transform the end-to-end constraints into a set of intermediate constraints on task attributes. In this section, we show how these constraints are derived, highlighting the extensions needed for distributed systems.

1. Constraints due to correlation, jitter and rate requirements

These constraints are handled in essentially the same way as in our uni-processor solution [9]. Correlation constraints are handled by creating a special sampler task responsible for sampling correlated input data. The correlation constraint is satisfied if the window of execution of the sampler task $D_s - O_s$ is bounded by the correlation requirement t_{cor}. In a distributed system scenario, the input sensors may be attached to different hosts. Therefore, we simply keep identical copies of the sampler task on each such host, running with identical values of T_s, O_s, D_s, and ϕ_s. The output jitter c onstraints are handled in the same way as for a single processor, and impose the following constraint on a task τ_i delivering external output Y: $T_i + (D_i - O_i) \le J_Y^u$ $T_i - (D_i - O_i) \ge J_Y^l$. The output rate constraints are even simpler, and simply impose bounds on the period of the output task τ_1, as follows: $T_Y^l \le T_1 \le T_Y^u$.

2. Constraints due to communication and freshness requirements

A crucial aspect of our approach is the communication scheme between producer-consumer pairs, and the delay introduced in transferring data from one task to another. A producer-consumer pair relationship exists between two tasks τ_p and τ_c, where τ_p writes to a data channel d_i, and τ_c reads from d_i. Similarly, it also exists between a task τ_i and a network task m_j, if τ_i writes to channel c_j, and between a network task m_i, and task τ_j, if τ_j reads from c_i.

In [9], we showed that if the producer and consumer periods are harmonically related, then a consumer can read fresh and correlated data from its input channels using virtual sequence numbers. For example, if T_p = 20 and = 60, T_c, τ_c needs only every third data item produced by T_p. Therefore, τ_p produces data items with sequence numbers $0, 1, 2, 3, 4, \ldots$, while τ_c simply reads the data items with sequence numbers 0,3,6, etc. Therefore, our first step is to add harmonicity constraint $T_c | T_p$ (T_c is exactly divisible by T_p), between every producer-consumer pair (τ_p, τ_c) in the task graph.

A freshness constraint $F(Y|X) = t_f$ forms a chain of producer-consumer pairs. As the data flows through the chain, a delay is encountered in the processing involved at each task, and in transferring data from one task to another. Furthermore, for smooth flow of data, we must ensure that a consumer never overtakes a producer-that is, whenever the consumer task instance is started, the corresponding producer task instance should have finished execution. Therefore, a freshness constraint imposes two types of constraints: one to establish proper precedence between producer consumer pairs, and another to ensure that the freshness bound is satisfied over the entire chain.

2.1. Freshness constraints derivation in single processor solution

Figure 2 illustrates the precedence scheme used between producer-consumer pairs in our single processor solution [9]. We note that harmonicity allows us to restrict attention to only the first instance of the consumer. As is evident from the figure, proper precedence is maintained if $\phi_p = \phi_c$ and $O_c \ge D_p$. Now, consider a freshness constraint $F(Y|X) = t_f$ with task τ_1 reading the input X , and τ_n producing the output Y. The data path from X to Y forms a chain of producer-consumer pairs, for which the end-to-end delay is bounded by $D_n - O_1$, and therefore the freshness constraint is satisfied if $D_n - O_1 \le t_f$.

On a single processor, the scheme can be optimized by deferring the precedence requirement to the scheduler. For instance a deadline monotonic scheduler would ensure precedence if we set $O_c = O_p$, and $D_c \ge D_p$. This brings in the advantage that the consumer is not artificially forced to wait until after the deadline of the producer, and also we avoid decomposing the end-to-end freshness bound into fixed slack for each task in the chain. This optimization is feasible for consumer tasks that do not have constraints on offsets (in our scheme, any task that does not read correlated input data, or produce output data with jitter constraints).

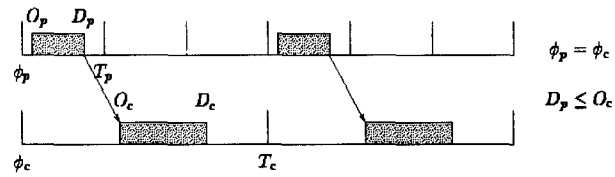

Fig. 2. Satisfying Precedence Requirements through Offsets.

2.2. Extending to distributed systems

When extending to distributed systems, we find that the above scheme is no longer desirable for freshness chains spanning multiple processors. The offset based scheme used for single-processor systems, brings in strong coupling between the freshness bounds and output task rates, since all tasks along a data chain must execute within the period of the output task. This coupling is undesirable when the freshness bounds are loose compared to output rate requirements, and may

easily make the system unschedulable. For example, consider the example ATG presented earlier (Figure 1). With this scheme, all of τ_1, m_1, and τ_3 must execute within τ_3's period, which is no more than $20ms$. Clearly, this is unnecessary, and would make the constraints trivially unsatisfiable if $e_1+e_1^m+e_3>20$, even though utilization on each processor (including network) may be well below 1. Note that this coupling is not harmful on a single processor, since if $e_1+e_1^m+e_3>20$, it implies that the processor utilization is greater than 1, and therefore the constraints cannot be satisfied.

To decouple the freshness bounds from task periods, we can explicitly use task phasing (ϕ's). This scheme is illustrated in Figure 3. As is evident from the figure, precedence is established if $\phi_p+D_p\leq\phi_c+O_c$. We note that this is a generalization of the offset scheme, and degenerates to the offset scheme whenever $\phi_p=\phi_c$. The end-to-end delay over a data chain is now given by $(\phi_n+D_n)-(\phi_1+O_1)$.

The introduction of phase attribute to tasks, adds another set of variables, and hence additional complexity for the constraint solver. Fortunately, we can reduce the number of phase variables by realizing that on a single processor, this generalization presents no significant advantage. Therefore, we may set $\phi_p=\phi_c$, whenever the producer and consumer are on the same processor, thereby reducing it to the scheme used in single processor solution. Furthermore, as in the single processor case, a further optimization may now be made by deferring the precedence to the scheduler - whenever it is feasible.

A special note must be made when the consumer is a network task, and the producer is a computation task. Recall from our model that network tasks have offset equal to 0. Furthermore, since each channel has a single writer, and we use virtual communication tasks to transfer data from the producer's buffers to the network, we may always trigger the virtual communication task at the deadline of the producer. Thus, we get $\phi_c=\phi_p+D_p$.

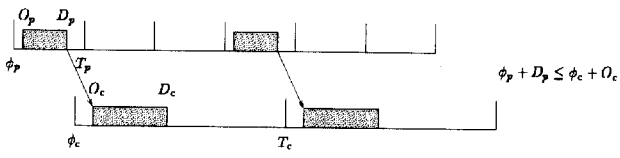

Fig. 3. Satisfying Precedence Requirements through Phasing.

Local Freshness Bounds. A freshness constraint $F(Y|X)=t_f$ may well stretch over several partitions $P_1,\ldots,P_k$, and network channels $c_1,\ldots,c_{k-1}$, forming a data path from system input X to system output Y. Let Γ_i denote the chain of tasks on each partition P_i, and $O(\Gamma_i)$ denote the offset of first task on the chain, $D(\Gamma_i)$ denote the deadline of the last task on the chain, and $\phi(\Gamma_i)$ denote the phase of the first task on the chain. From the preceding discussion, we can derive the maximum delay along different components as follows: (i) For each partition P_i, the maximum delay is given by $D(\Gamma_i)-O(\Gamma_i)$, (ii) for each channel (c_i), since the channel has only one (network) task (m_i), the maximum delay is simply D_i^m, and (iii) the maximum delay in transferring data from a channel c_{i-1} to a partition P_i is given by $(\phi(\Gamma_i)+O(\Gamma_i))-(\phi_{i-1}^m+D_{i-1}^m)$.

The end-to-end freshness constraint is satisfied if the sum of all the delay components is no more than t_f. However, writing the constraint in this form poses significant problems for the constraint solver, since it merges two distinct sub-problems: (1) decomposition of end-to-end freshness bounds into a local bound for each delay component, and (2) derivation of offsets and deadlines to satisfy the local bounds (and other constraints). This coupling is undesirable as it complicates the derivation of deadlines and offsets and makes it difficult to incorporate suitable schedulability criteria when doing so. Therefore, we introduce two new variables (termed "freshness" and "delay") - one for each local freshness bound, thereby effectively decoupling the two sub-problems.

The freshness variables are used to bound the delay on each partition and channel, and are labeled F_1, F_2, etc. The delay variables are used to bound the delay in transferring data between partition and channels, and are labeled δ_1, δ_2, etc. The reason for distinguishing them is simple: the two types of variables have different interpretation and therefore must be solved using different objective criteria. The freshness variables involve processing delay from tasks, and therefore must have values that can accommodate the execution of tasks. Therefore, consideration must be paid to relative loads on the processors when assigning a value to them. On the other hand, delay variables represent pure delay, and therefore, the same criteria cannot be used.

3. Deriving constraints for example ATG.

We revert our attention back to the example system presented earlier. We first introduce new variables F_1, F_2, F_3, F_4 for the delay on partitions, and F_5 and F_6 for delay on channels. We also use δ_1 and δ_2 as two delay variables. In addition to the intermediate constraints derived from the end-to-end requirements, we also have constraints due to execution time requirements, and the periodic tasking model itself. Therefore, for each task we have : $D_i-O_i\geq e_i, O_i\geq 0$, and $D_i\leq T_i$. Before the constraints are presented to the constraint solver, many simplifications may be made as noted below. The final set of constraints given in Table 1 reflects the constraint set after the simplifications.

(1) Since each channel has a single sender, we can simply set the period of a network task to its corresponding producer. Therefore, we get $T_1^m=T_1$, and $T_2^m=T_2$. Likewise, we can set the phase of a network task to coincide with the deadline of the sender, and therefore we can set $\phi_1^m=\phi_1+D_1$, and $\phi_2^m=\phi_2+D_2$.

(2) Since there are no input correlation or output jitter constraints, we do not need task offsets; therefore, we can simply set $O_1 = O_2 = O_3 = O_4 = 0$.

(3) For each of the tasks which read external input, we can simply set the phase to 0; therefore we get $\phi_1 = \phi_2 = 0$.

(4) We can also establish the phase of task τ_4, since it is only dependent on the network task m_2; and thus we get $\phi_4 = \phi_2^m + D_2^m = \phi_2 + D_2 + D_2^m = D_2 + D_2^m$

(5) Since we know that on each partition (or channel) the maximum delay is bounded by the output task's period, we add the following constraints: $F_1 \le T_1$, $F_2 \le T_3$, $F_3 \le T_3$, $F_4 \le T_4$, $F_5 \le T_1^m$ and $F_6 \le T_2^m$. These constraints are helpful in the derivation of local freshness bounds.

Table 1. Intermediate constraints for example ATG.

Harmonicity	$T_3 \mid T_1^m$, $T_3 \mid T_2^m$, $T_4 \mid T_2^m$, $T_1^m = T_1$, $T_2^m = T_2$
Precedence	$\phi_4 \ge D_1 + D_1^m$, $\phi_3 \ge D_2 + D_2^m$
End-to-End Freshness	$F_1 + F_3 + \delta_1 + F_3 \le 64$, $F_2 + F_6 + \delta_2 + F_3 \le 54$, $F_2 + F_6 + F_4 \le 85$
Local Freshness, and Delay	$D_1 \le F_1$, $D_2 \le F_2$, $D_3 \le F_3$, $D_4 \le F_4$, $D_1^m \le F_5$, $D_2^m \le F_6$, $\phi_3 - D_1 - D_1^m \le \delta_1$, $\phi_3 - D_2 - D_2^m \le \delta_2$, $F_1 \le T_1$, $F_2 \le T_2$, $F_3 \le T_3$, $F_4 \le T_4$, $F_5 \le T_1^m$, $F_6 \le T_2^m$
Rate	$T_3 \le 20$, $T_4 \le 71$
Execution	$7 \le D_1$, $8 \le D_2$, $6 \le D_3$, $21 \le D_4$, $7 \le D_1^m$, $9 \le D_2^m$

V. Constraint solver

Once we have expressed the end-to-end constraints as a system of intermediate constraints, the constraint solver is invoked to generate instantiations for the periods, deadlines, offsets, and phases, as well as freshness and delay variables. Clearly, any solution that satisfies the intermediate constraints preserves correctness ; that is if the final task set is schedulable then end-to-end constraints will be satisfied. The constraint solver must also incorporate notions of schedulability in generating the solutions to avoid generating trivially unschedulable solutions. Therefore, we solve the constraints using objective functions which capture the notion of schedulability.

We highlight the problems faced in extending the solution for distributed systems, and then present the new results.

1. Extending the constraint solver for distributed systems

Additional complexity is introduced into the constraint solving process, as we move from a single processor system to a distributed system. The added complexity arises from several sources:

- *End-to-End Freshness*: A major source of additional complexity arises from end-to-end freshness requirements, which resulted in the introduction of new variables. In order to solve for these new variables, we must define and solve additional sub-problems, and integrate them into the overall scheme.
- *Schedulability*: Incorporating schedulability notions in the constraint solving process for a distributed system environment is significantly harder than for a single processor system. To ensure schedulability of the entire system, we have to ensure that proper load balancing is achieved, not only at the macro level of overall utilization, but also on the micro level of individual deadlines.
- *Size*: In a complex distributed system, the number of tasks (and therefore, the number of variables and constraints) can be quite large. The constraint solver must scale well to the increased complexity due to size.

In extending the solution, we build upon the solution for single processor systems, retaining the essential structure of the approach. We recall that the freshness variables were introduced to avoid coupling between task variables on each host. Therefore, our first intuition was to solve for local-freshness bounds first, and then proceed with solving the constraints for each processor in much the same way as the single processor case. However, we soon realized that this approach was not very effective due to two factors: (i) in the absence of any knowledge about load on each processor[2], no good objective function could be determined for solving for freshness bounds that would capture the notion of schedulability on each processor; and (ii) the values of freshness bounds may easily result in a null feasible space for periods due to non-linear harmonicty constraints. Therefore, as in the single processor solution, we had to go back to addressing the period-assignment problem first. Once, the periods are known, the freshness bounds may be derived, utilizing the knowledge about relative load on each processor. Once the local freshness bounds are determined, we can solve for the deadlines and offsets, maximizing the window of execution for each task, and finally determine a feasible phasing for each of the tasks. The overall approach is outlined below:

1) The first sub-problem that we solve is the period assignment. As in our single-processor case, we first reduce the constraint set to one involving periods only using Fourier variable elimination. However, unlike the single processor case, we do not minimize overall utilization; instead we assume that maximum utilization bounds are known for each processor. The utilization bounds may be viewed as estimates of available utilization for this ATG on a given processor. Any solution that satisfies these bounds is accepted.

2) Once the periods are known, we proceed to solve the sub-problem of decomposing end-to-end freshness bounds into local freshness bounds. Variable elimination is again used to restrict attention to freshness variables. The schedulability notions are captured in the objective function by making use of information about utilization of each task, each processor, and the available utilization on each processor.

3) Once the periods and the freshness bounds are found, we are still left with a system of constraints

2) We use the term processor to indicate both CPU and network resources.

involving deadlines, offsets, and phases for the tasks. These constraints largely involve variables on a single partition, but some interdependencies may arise due to network communication. We first solve for offsets and deadlines, with the objective of maximizing the window of execution of each task, just as in our single processor solution. However, each processor is handled in a serial manner due to the interdependencies. In order to maximize schedulability, the ordering is determined by the utilization available, and the actual utilization of the task set on each processor.

4) Finally, feasible task phasings are determined by traversing the task graph in a topological sort order, and minimizing the phase of each task.

In the remaining part of this section, we delve into the extensions needed in the constraint solver, specifically in determining local freshness bounds.

2. Period assignment

The extensions to period-assignment are straightforward, and therefore we simply illustrate the algorithm on the example ATG. The constraints that we derive after eliminating non-period variables are given below:

$$T_1^m = T_1,\quad T_2^m = T_2,\quad T_3 | T_1^m,\quad T_3 | T_2^m,\quad T_4 | T_2^m$$
$$T_1^m \geq 7,\quad T_2^m \geq 9,\quad T_1 \geq 5,\quad T_2 \geq 8,\quad 6 \leq T_3 \leq 20,$$
$$21 \leq T_4 \leq 71$$

We assume that the maximum utilizations are given as:

$$U_1^{\max} = 0.95\quad U_2^{\max} = 0.90\quad U_N^{\max} = 0.82$$

From the harmonicity requirements, it follows that $T_1 = T_1^m = GCD(T_3) = T_3$, and $T_2 = T_2^m = GCD(T_3, T_4)$. Therefore, a feasible solution (in fact the only one) that satisfies the utilization bounds is $T_3 = 20$, $T_4 = 60$, and $T_1 = T_1^m = T_2 = T_2^m = 20$.

3. Deriving local freshness bounds

The constraints for this sub-problem are obtained by instantiating the period values, and projecting the constraint set onto the sub-space of freshness variables. Let $F_1, F_2, \ldots, F_n$ be the variables denoting local freshness bound for a chain of tasks on some data path, allocated to a single processor. The constraint set includes a lower bound on each F_i (denoted as F_i^l), an upper bound on each F_i (denoted as F_i^u), and a set C of end-to-end freshness constraints. The lower bounds arise from execution time requirements, while the upper bounds arise from the task periods[3]. Reverting back to the example, we get the following end-to-end constraints:

$$F_1 + F_5 + F_3 \leq 64,\quad F_2 + F_6 + F_3 \leq 54,$$
$$F_2 + F_6 + F_4 \leq 85$$

Note that at this stage the delay variables δ_i's have been eliminated as well. In addition the lower bounds are given as:

$F_1^l = 5, F_2^l = 8, F_3^l = 6, F_4^l = 21, F_5^l = 7, F_6^l = 9$, while the upper bounds are given as: $F_1^u = F_2^u = F_3^u = F_5^u = F_6^u = 20$, and $F_4^u = 60$.

The main challenge in the problem arises from addressing notions of schedulability. Therefore, our first step is to define an appropriate cost function $\rho(F_1, F_2, \ldots, F_n)$, which when minimized, maximizes the schedulability across the distributed system.

Deriving a Cost Function. Let $\rho_i(F_i)$ be some cost function that captures schedulability of the chain of tasks Γ_i represented by F_i. Recall that F_i denotes a bound on the total time available to execute the tasks in Γ_i. Therefore as F_i is increased, the schedulability increases for tasks in Γ_i. Hence, $\rho_i(F_i)$ must be a function that decreases as F_i increases. Unfortunately, while increasing F_i may increase the schedulability of the chain; it may actually decrease the schedulability of the entire system by over constraining another chain represented by a variable F_j. Since the overall system is schedulable if and only if each of its components is schedulable, the overall cost is best captured as follows:

$$\rho(F_1, F_2, \ldots, F_n) = \max_i \rho(F_i)$$

with the schedulability maximized when ρ is minimized. We are still left with the problem of defining each ρ_i. The approach that we take directly addresses the schedulability notion. Recall that each F_i represents a bound on the difference between the deadline of the last task of the chain, and the offset of the first task of the chain. In other words, it presents a bound on the maximum "response time" for the chain of tasks. Therefore, in addressing schedulability, we must tie the value of F_i to the response time of the chain. Let us assume that for each F_i, we have an estimate of the response time (given as R_i) for the chain of tasks Γ_i that it represents. The expected response time would lie somewhere in the range $[F_i^l, F_i^u]$. If we set $F_i = R_i$ for all variables, and assuming that the estimated response times are accurate, then all chains of tasks will exactly satisfy their local delay bound. We postulate a cost function given as $\rho(F_i) = R_i/F_i$, which achieves a value of 1 for $F_i = R_i$. The cost function ρ may be viewed as the normalized speed of the processor at which the expected response time equals the value of F_i. For instance, when $F_i = R_i/2$, then the expected response time would equal F_i on a processor that is twice as fast, $(\rho = 2)$. Likewise, when $F_i = 3R_i$, a processor that is three times as slow $(\rho = 1/3)$ would result in making the expected response time equal to F_i.

Estimating Response Times. Consider a single processor with the set of tasks $\tau_1, \tau_2, \ldots, \tau_m$ allocated on that processor from the ATG. The utilization of each task τ_i is known at this time, and given by e_i/T_i. Let U be the sum of the utilization for all the tasks, and let $U^{\max}$ denote the utilization available for the tasks. The response times are estimated conservatively assuming an idealized processor sharing scheduling discipline, in which each task is given a fraction of the processor at each

3) Recall that on each processor, the freshness bound is limited by the output task's period.

time instant. Thus, a task with a fraction f ($0 < f \leq 1$) of the processor can be thought of running alone on a processor which is $1/f$ times slower. Therefore, if e is the normal execution time of the task, it takes e/f time to complete the task, which we estimate as its response time.

In our scheme, $U^{\max}$ gives the fraction of the processor available to the entire ATG. Dividing this amount proportionately among all tasks, the fraction available to task τ_i is given as:

$$f_i = \frac{U_i}{U} * U^{\max}$$

and the response time of task τ_i is given as e_i/f_i. The response time of a chain of tasks, is now simply estimated as the sum of response time of the tasks in the chain.

Solving the Constraints to Minimize Cost. The price that we pay for a good objective function is the introduction of non-linearity, and therefore the constraints can no longer by solved using linear programming methods. Fortunately, the nature of the constraint set, and the objective function lend to a simple algorithm that minimizes the overall cost, which is presented in Figure 4.

The algorithm begins by transforming the original end-to-end constraints C by substituting each F_i with R_i/ρ_i. The new constraint set $\hat{C}$ has variables ρ_i, which directly reflect the cost. Therefore, the objective function is to simply minimize $\max_i \rho_i$.

We also establish lower and upper bounds on each ρ_i as follows: $\rho_i^l = R_i/F_i^u$, and $\rho_i^u = R_i/F_i^l$. We first check if $\hat{C}$ is satisfiable when $\rho_i = \rho_i^u$ for each i, since if it is not then no solution exists and we can terminate. On the other hand, if $\hat{C}$ is satisfiable when $\rho_i = \rho_i^l$, we can also terminate since this is the best possible solution, with a cost of $\max_i \rho_i^l$. In general, an optimal solution occurs when each ρ_i takes some value in the range $[\rho_i^l, \rho_i^u]$. Clearly, the optimal cost lies in the interval $[\max_i \rho_i^l, \max_i \rho_j^u]$.

To find an optimal solution, we rely on the following fact: If the optimal cost ρ^* is less than $\min_i(\rho_i^u)$, then an optimal solution is simply $\rho_i = \rho^*$, and may be obtained by substituting each ρ_i with a new variable ρ, and then minimizing the value of ρ. A complication arises when the optimal cost is greater than $\min_i(\rho_i^u)$. In this case, an optimal solution is given by: $\rho_i = \min(\rho_i^u, \rho^*)$. The two cases can be combined by ignoring the upper-bounds on ρ_i when finding the smallest value of ρ that satisfies the constraints. If the value so obtained is no more than $\min_i(\rho_i^u)$, then we have an optimal solution. On the other hand, if the value so obtained is greater than $\min_i(\rho_i^u)$ then some of the upper bounds are violated. For each of these ρ_i's, the optimal value is simply ρ_i^u. We can now substitute these values into the constraints and repeat the procedure until done.

A further optimization may be done when $\rho^* < \min_i(\rho_i^u)$. While the overall cost may not be reduced any further, the values of individual ρ_i's may still be reduced. This is true for any ρ_i that is not in a saturated constraint (i.e., constraints that would be violated if any ρ_i was reduced any further), when all ρ_i's are substituted by ρ^*. Therefore, for each constraint that is saturated, we establish the value of each variable in it as ρ^*, and substitute these values in the constraint set. The remaining constraint set with a reduced set of variables may now be solved again using the same procedure.

The algorithm presented in Figure 4 implements the above ideas. When the algorithm terminates, the optimal cost for each F_i is given by ρ_i, from which F_i is simply obtained as $F_i = R_i/\rho_i$.

Algorithm Freshness-Bound Decomposition Input:

F: set of variables $F_1, F_2, \ldots, F_n$, Ψ: set of variables $\rho_1, \rho_2, \ldots, \rho_n$.

$\hat{C}_l$: set of constraints $\rho_i \geq \rho_i^l$, $\hat{C}_u$: set of constraints $\rho_i \leq \rho_i^u$.

$\hat{C}$: set of end-to-end freshness constraints.

1 **if** not consistent ($\hat{C}[\forall i: \rho_i/\rho_i^u]$) **then return fail**;
2. **if** consistent ($\hat{C}[\forall i: \rho_i/\rho_i^l]$) **then** $\forall i$: $\rho_i := \rho_i^l$; **return success**;
3. Let $C' = \hat{C}[\forall i: \rho_i/\rho] \cup \hat{C}_l[\forall i: \rho_i/\rho]$, and
4. Let ρ^* be the minimum value of ρ that satisfies C'.
5. **if** $\rho^* > \min(\rho_i^u)$ **then**
6. **foreach** i **do**
7. **if** $\rho^* > \rho_i^u$ **then**
8. $\rho_i := \rho_i^u$; $F_i := R_i/\rho_i$;
9. $\hat{C} := \hat{C}[\rho_i/\rho_i^u]$; $\Psi := \Psi - \rho_i$;
10. **else**
11. Let C'' be the set of inequalities in C' that are "just satisfied" at $\rho = \rho^*$.
12. **foreach** x in C'' **do**
13. **foreach** variable ρ_j in x **do**
14. $\rho_j := \rho^*$; $F_j := R_j/\rho^*$;
15. $\hat{C} := \hat{C}[\rho_j/\rho^*]$; $\Psi := \Psi - \rho_j$;
16. **endif**
17. **if** not Nil Ψ **then goto 3**.

Fig. 4. Decomposition of End-to-End freshness bounds.

Deriving Freshness Bounds for Example ATG. The first step is to derive the estimated response times, and the following values are obtained.

R_1	R_2	R_3	R_4	R_5	R_6
13.68	13.68	14.44	43.33	19.51	19.51

From the bounds on F_i, we obtain the lower and upper bounds on ρ_i as follows:

	ρ_1	ρ_2	ρ_3	ρ_4	ρ_5	ρ_6
ρ_i^l	0.68	0.68	0.72	0.72	0.97	0.97
ρ_i^u	2.74	1.71	2.41	2.06	2.78	2.17

The optimal solution is found as follows:

1. In the first round, the minimal value of ρ is found to be 0.97 which is larger than $\min \rho_i^u = 1.71$. Therefore, $\rho^* = 0.97$. The cost cannot be reduced any further since $\rho_5^l = \rho_6^l = 0.97$. Thus, we obtain $F_5 = F_6 = 20$, and continue to reduce ρ_i for other chains. The reduced end-to-end constraints are now: $F_1 + F_3 \leq 44$, $F_2 + F_3 \leq 34$, and $F_2 + F_4 \leq 65$.
2. In the second round, the minimal value of ρ is found to be 0.87, below which the constraint $F_2 + F_4 \leq 65$ becomes violated. Therefore, we set $F_2 = 15$ and $F_4 = 49$ [4], and simplify the constraints to $F_1 + F_3 \leq 44$, and $F_3 \leq 19$.
3. In the third round, $F_3 \leq 19$ is saturated, and we set $F_3 = 19$. Finally, we set $F_1 = 20$.

The final solution for freshness bounds is given below:

F_1	F_2	F_3	F_4	F_5	F_6
20	15	19	49	20	20

Solving for Delay Variables: After the values of freshness variables have been determined, we are still left with delay variables. The previous algorithm distributes the end-to-end delay bounds as much as possible for each partition and channel. Therefore, any slack left may be given to the delay variables. We now have the following constraints: $0 \leq \delta_1 \leq 5$ and $0 \leq \delta_2 \leq 0$. Therefore, we give a value of 0 to δ_2 and 5 to δ_1.

4. Deriving offsets and deadlines and phase

The last stage of our constraint solving process is the derivation of task deadlines, offsets, and phases. The objective here is to maximize the window of execution of each task. In our single processor solution, we presented a simple greedy approach which achieved this by maximizing the deadlines and minimizing the offsets, one at a time. However, a greedy approach like this is not entirely satisfactory, since maximizing one deadline may over-constrain another one. Here we sketch a similar, but slightly improved algorithm, which combines the greedy approach with the ideas used for freshness bounds.

1) We solve the constraints by focusing on the variables of one partition at a time. Again, we use variable elimination to restrict the constraint set to variables of a single partition. For choosing the order of partition, we define the relative load of a processor as $U/U^{\max}$, where U is the total utilization of the tasks belonging to the processor, and $U^{\max}$ is the utilization bound on the processor. We then visit the variables of the partitions in the decreasing order of the relative load, the idea being that a processor with high relative load has the least amount of slack, and therefore, the windows of tasks residing on that processor must be optimized first.

2) For each partition, we begin with minimizing the offset of the input tasks, one by one. Most often, the offset of input tasks is used only for correlation constraints, and there is little benefit in having a non-zero offset.

3) The input offsets, once determined give us the starting points for windows of all tasks, except perhaps the output tasks (see the discussion on precedence due to freshness requirements). Therefore, we proceed to maximize the deadlines. Note that the problem here is very similar to the problem of finding freshness bounds, and we use the same approach based on estimated response times. Once the deadlines have been determined, the end-points of all offsets are known, and we solve for the offsets of the output tasks.

4) After all offset and deadline variables are solved, for all the partitions, we simply minimize the phase variables - one at a time.

Revisiting our example, we find that the relative loads are given by: 0.684, 0.722 for the two hosts, and 0.975 for the network. Therefore, we start with the network, and solve for D_1^m, and D_2^m. In this case, the solution is trivial and we set $D_1^m = D_2^m = 20$. We then move on to host 2, with variables D_3 and D_4. Again, the optimal values of these two are easily found to be $D_3 = 19$ and $D_4 = 49$. The remaining constraints are now given as:

$D_1 \leq 20$, $D_2 \leq 15$, $5 \leq D_1$, $8 \leq D_2$, $D_1 \leq D_2, D_2 \leq D_1 + 5$

The first two constraints are simply the upper bounds on the deadlines arising from the local freshness bounds. Likewise, the next two constraints are simply the lower bounds from the execution time requirements. The final two constraint arise from the interactions of the two data chains from inputs X_1 and X_2 to output Y_1. Recall that we have earlier solved for δ_1 and δ_2, and found values of 5 and 0 respectively. From the above constraints, we obtain the largest values of D_1 and D_2 as 15 and 20 respectively. Finally, we solve for ϕ_3 and find that its minimal (in fact the only value) is 35. The final solution is given below:

	T	ϕ	O	D		T	ϕ	O	D
τ_1	20	0	0	15	τ_2	20	0	0	20
τ_1^m	20	15	0	20	τ_2^m	20	20	0	20
τ_1	20	35	0	19	τ_2	60	40	0	49

4) We restrict the solution to integers, and therefore take the floor of the actual value.

VI. Buffer allocation

Non-blocking accesses to channels lies at the heart of our resource management scheme. In [9] we present a solution to data channel implementation in which we start with an unlimited buffer assumption, and then bound the size of the end result. Unfortunately, where hardware limitations are present, such a straightforward solution may fail to achieve the desired asynchrony. In this section, we show how network channels can be implemented, using a specific network controller for Controller Area Network (CAN) described in [21]. We first review the data channel management from our original single processor solution, and then present the network channel management.

1. Data channel management

The most pressing requirement in channel management is to ensure that a consumer task always sees fresh and correlated data items in the data channels of its producers. The harmonicity requirement greatly helps us meet the requirement, since we can consider only one period of the consumer, as shown in [9].

Let τ_p and τ_c denote a producer and its consumers, respectively. Due to enforcement of harmonicity on a data chain, τ_c only reads a data item that was generated within its current period. Thus, so long as τ_p keeps at least n_i data items where $n_i = \frac{T_c}{T_p}$, τ_c can always find a correct data item. This is true for all consumers of d. Therefore, the size s of d is chosen as the maximum of n_i. Once a value is picked for s, the next data item can be found using the following index operation.

$$index_i = (index_i + n_i) \bmod s$$

where $index_i$ is initially set 0 for all i.

2. Network channel management

The reason for choosing CAN network is that it provides several benefits: First, it provides for a broadcast bus, which is desirable to implement multiple readers in an ATG. Second, it allows for simple mechanical arbitration solely relying on static priority scheme; this in turn enables us to use a relatively simple schedulability test, which has the same theoretical basis as a class of processor schedulability analyses discussed in [20]. Thus, uniform treatment between processor and network scheduling is possible.

As in [21], we consider Intel 82527 network controller as a network interface between a host CPU and a CAN bus. It seems appropriate to assume a specific controller, since different controllers have different types of limitations which may affect the system calibration process of our approach.

For example, the Intel 82527 controller has only 15 slots for incoming and outgoing messages, while an ideal network controller for CAN would give 2048 slots[5] for messages so that a message to be sent can be simply put into a slot corresponding to the message identifier. This invalidates the unbounded channel size assumption and leads to task blocking, since a task has to wait before it writes a message into a slot until the other message in the slot is transmitted. To avoid such blocking, we either ensure that a slot is always emptied before next write, or provide secondary buffer spaces so that messages are temporarily kept. Since the former approach requires global scheduling of both the network and host processors, we choose the latter approach. Before we proceed, we make the following assumptions: (1) Message identifiers are pre-determined when an ATG is built such that each network channel has a unique identifier across the ATG; (2) Slots in Intel 82527 are programmed to either receive, or send messages; and (3) Each message is mapped into a unique slot in a transmit host and in each of receive hosts, as described in [21].

We use an interrupt-based handshaking implementation in which each slot is independently programmed to generate an interrupt upon either receiving or sending a message. The key component of the approach is a priority-ordered queue which is created for, and associated with each slot used for transmission. The details of transmit slot implementation are itemized as follows.

- For each transmit slot, we create a priority-ordered queue in the transmit station.
- When a task attempts to write a message into a corresponding slot, it checks if the slot is empty. If so, it directly writes the message into the dual-ported slot; otherwise, it puts the message into the corresponding priority-ordered queue. As the priority-ordered queue may be written by multiple tasks and read by an interrupt handler, accesses to the queue are protected by mutual exclusion.
- When a message is sent to the network, an interrupt is issued and then a handler is dispatched. The handler moves a message at the top of the corresponding priority-ordered queue into the slot.

For the purpose of scheduling, we need to compute the maximum interference that a task can get due to interrupts raised to move messages from the priority-ordered queue to the corresponding slot. Suppose that a host has k transmit channels. Since each channel has its own message identifier, we have a message set $M = \{m_1, m_2, \ldots, m_k\}$ for the host.

The maximum number of transmit interrupts (I_i) task τ_i running on the node can see is:

$$I_i = \sum_{j=1}^{k} \left(\left\lceil \frac{D_i}{T_j^m} \right\rceil + 1 \right).^{6)}$$

Thus the maximum interference from these interrupts is simply: $t_i^{tx} = I_i \cdot t^{intr}$, where t_{intr} be the maximum transmit interrupt handling latency.

5) In reality, CAN allows for only 2032 message identifiers.

6) Note that the deadline of a task or a message is no greater than the period in the derived constraints.

The receive channel implementation is similar and is summarized below:

- For each message identifier mapped into a receive slot, create a FIFO queue in the receive host.
- When a message is delivered into a receive slot from the network, an interrupt is issued and an associated handler is dispatched.
- The interrupt handler moves the message into the corresponding FIFO queue.

We can compute the receive interrupt interference in exactly the same way. Let $M=\{m_1, m_2, \ldots, m_l\}$ be a set of messages that a host receives from the network. The maximum number of interrupts (denoted I_i) that task τ_i on the host can get is:

$$I_i = \sum_{j=1}^{l} \left(\left\lceil \frac{D_i}{T_j^m} \right\rceil + 1 \right).$$

Virtual Communication Tasks. As mentioned earlier, there exist virtual communication tasks to move messages from sender tasks to their corresponding slots (or priority-ordered queues). We implement such tasks as call-out interrupt handlers which are called at the deadlines of sender tasks, since interrupt-based implementation of virtual communication task results in shorter context switching latency than task-based implementation. Here, we account for the interference of such interrupts.

Suppose that a host has n running tasks, among which k tasks generate network messages. Let $\{\tau_1, \tau_2, \ldots, \tau_k\}$ denote those tasks. The maximum number of interrupts (denoted I_i) that a task τ_i on the host can get is:

$$I_i = \sum_{j=1}^{k} \left\lceil \frac{T_i}{T_j} \right\rceil .$$

VII. Conclusion

We have presented a resource conscious methodology for designing distributed real-time systems as an extension of our original approach for single processor real-time systems [8][9]. This methodology enables programmers to streamline the end-to-end design of real-time systems by way of semi-automatic tool-based approach. We believe that a tool developed using the ideas developed in the paper will be very useful for rapid prototyping of designs, and in identifying and eliminating bottlenecks.

The solution presented in this paper uses the overall structure and principles of the single processor solution. However, the complexity inherent in distributed real-time systems resulted in significant extensions to the single processor case. The overall approach may be viewed in several stages. First the end-to-end constraints are mapped to a set of intermediate constraints on task attributes. The intermediate constraints are then solved using heuristics to achieve schedulability. The constraint solver itself works in many stages - solving a well-specified sub-problem at each stage. This decomposition of the constraint solver helps in managing its complexity, and allows it to scale to large systems. Furthermore, it is also helpful in tuning each set of variables according to different objective criteria, as well as identifying potential bottlenecks in the system. Finally, once the task attributes are determined by the constraint solver, we automatically synthesize the data and network channels, and account for their memory requirements and interrupt overheads.

There exist many directions along which our approach can be extended. The most pressing demand on this methodology is validating the approach through applying it to real system development. Since this in turn requires the implementation of a tool, we are closely working with the TimeWare project group at University of Maryland in developing a tool for the approach. A preliminary version of the tool has already been developed for single processor environments. We hope that the tool will help us in addressing realistic applications, and help tune and extend the approach. We are also investigating extensions of our approach for multimedia applications, which need more flexible (and perhaps) blocking buffer management.

References

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

[2] R. Bettati and J. W.-S. Liu, "End-to-end scheduling to meet deadlines in distributed systems," *Proceedings of IEEE Conference on Distributed Computing Systems*, pp. 452-459, 1992.

[3] A. Burns, "Preemptive priority based scheduling: An appropriate engineering approach," In S. Son, editor, *Principles of Real-Time Systems*. Prentice Hall, 1994.

[4] B. Chen, G. Agrawal and W. Zhao, "Optimal synchronous capacity allocation for hard real-time communicatins with the timed token protocol," *Proceedings of IEEE Real-Time Systems Symposium*, pp. 198-207, December, 1992.

[5] G. Dantzig and B. Eaves, "Fourier-Motzkin elimination and its dual," *Journal of Combinatorial Theory (A)*, 14:288-297, 1973.

[6] D. Ferrari and D. C. Verma, "Scheme for real-time channel establishment in wide-area networks," *IEEE Journal on Selected Areas in Communications*, 8(3):368-379, 1990.

[7] J. Garcia and M. G. Harbour, "Optimized priority assignment for tasks and messages in distributed hard real-time system," *Proceedings of IEEE Workshop on Parallel and Distributed Real-Time Systems*, 1995.

[8] R. Gerber, S. Hong and M. Saksena, "Guaranteeing

end-to-end timing constraints by calibrating intermediate processes," *Proceedings of IEEE Real-Time Systems Symposium*, pp. 192-203. *IEEE Computer Society Press*, December, 1994

[9] R. Gerber, S. Hong and M. Saksena, "Guaranteeing real-time requirements with resource-based calibration of periodic processes," *IEEE Transactions on Softpware Engineering*, 21(7), July, 1995.

[10] H. Gomaa, "A software design method for real-time systems," *Communication of the ACM*, 8(2):938-949, September, 1984.

[11] J. Sun, J. Liu and R. Bettati, "An end-to-end approach to scheduling periodic tasks with shared resources in multiprocessor systems," Technical report, *Department of Computer Science*, University of Illinois, 1994. Unpublished Report.

[12] D. D. Kandlur, K. G. Shin and D. Ferrari, "Real-Time communication in multi-hop networks," *Proceedings of IEEE Conference on Distributed Computing Systems*, pp. 300-307, May, 1991.

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

[14] 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.

[15] S. Mukherjee, D. Saha, M. Saksena and S. K. Tripathi, "A Bandwidth allocation scheme for time constrained message transmission on a slotted ring LAN," *Proceedings of IEEE Real-Time Systems Symposium*, December, 1993.

[16] M. Di Natale and J. Stankovic, "Dynamic end-to-end guarantees in distributed real-time systems," *Proceedings of IEEE Real-Time Systems Symposium*, pp. 216-227, 1994.

[17] K. Ramamritham and J. A. Stankovic, "Scheduling algorithms and operating systems support for of real-time systems," *Proceedings of the IEEE*, 82(1):55-67, January, 1994.

[18] D. Saha, M. Saksena, S. Mukherjee and S. K. Tripathi, "On Guaranteed Delivery of Time-Critical Messages in DQDB," *Proceedings of IEEE Infocomm*, June, 1994.

[19] L. Sha and S. S. Sathaye, "A systematic approach to designing distributed real-time systems," *IEEE Computer*, 26(9):68-78, September, 1993.

[20] 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.

[21] K. Tindell, H. Hansson and A. Wellings, "Analysing real-time communications: Controller area network (can)," *Proceedings of IEEE Real-Time Systems Symposium*, December, 1994.

[22] K. W. Tindell, A. Burns and A. J. Wellings, "Guaranteeing hard real time end-to-end communications deadlines," Technical Report RTSRG/91/107, University of York, Department of Computer Science, December, 1991.

[23] S. H. Hong, "Scheduling algorithm of data sampling times in the integrated communcation and control systems," *IEEE Transactions on Control Systems Technologies*, 3(2):225-230, June, 1995.

Appendix A. Fourier variable elimination

Fourier variable elimination [5] works on system of linear constraints, and may be viewed geometrically as the projection of an n-dimensional polytope (described by the constraints) onto its lower-dimension shadow. The procedure is best illustrated through an example. Consider the constraints on variable ϕ_3 in our example ATG, which are given below:

$$D_1 + D_1^m \le \phi_3 \qquad \phi_3 \le D_1 + D_1^m + \delta_1$$
$$D_2 + D_2^m \le \phi_3 \qquad \phi_3 \le D_2 + D_2^m + \delta_2$$

This constraint set is solvable only if the following are satisfied:

$$0 \le \delta_1 \qquad D_1 + D_1^m \le D_2 + D_2^m + \delta_2$$
$$0 \le \delta_2 \qquad D_2 + D_2^m \le D_1 + D_1^m + \delta_1$$

This new set of constraints eliminates ϕ_3 by simply combining each lower-bound, with each upper-bound.

홍 성 수

1963년 10월 11일생, 1986년 서울대학교 컴퓨터공학과 졸업. 1988년 동대학원 석사. 1994년 Unitersity of Maryland at College Park, Dept. of Computer Science 박사. 1988년 2월 ~ 1989년 7월 한국전자통신연구소 연구원. 1994년 12월 ~ 1995년 3월 Fuculty Research Associate (Univ. of Maryland). 1995년 4월 ~ 1995년 8월 Member of Technical Staff Silicon Graphics Inc. 현재 서울대학교 공과대학 전기공학부 조교수. 관심분야는 실시간 시스템, 분산제어 시스템, 소프트웨어 엔지니어링, 오퍼레이팅 시스템 등.