Practical Considerations in Designing Distributed Real-Time Systems: A Case Study on an FIP-based System*

Minsoo Ryu and Seongsoo Hong

School of Electrical Engineering and Computer Science
Seoul National University
Seoul 151-742, Korea.
Email: msryu,sshong@redwood.snu.ac.kr

Abstract

In this paper we describe our case study of designing an FIP-based distributed numerical control system, focusing on practically validating the end-to-end design methodology that we have developed in [10, 11]. The design methodology was used to automatically derive task attributes such as periods, deadlines, and phases, from the high-level design of the numerical control system. Since the design methodology was developed under a number of simplifying and generalizing assumptions, we had to customize it during the case study, taking into account practical issues such as the clock synchronization and the network scheduling of FIP. In doing so, we also developed a new and improved deadline decomposition algorithm to increase the chances of producing schedulable timing constraints.

We have fully implemented the distributed numerical control system on top of the ARX real-time operating system that we have built, and performed extensive experiments by generating a controlled series of dozens of numerical controllers. These experiments clearly show that the proposed deadline decomposition algorithm outperforms existing ones, and that the end-to-end design method can be effectively applied to the design of highly predictable industrial real-time systems, if it is properly customized for the given hardware and software platform.

1 Introduction

Designing a distributed real-time system for factory and process automation is a multi-dimensional and multi-facet activity which involves various technical difficulties. The sources of such difficulties include, among many others, complexity in (1) task decomposition and allocation, (2) timing constraint derivation, and (3) distributed real-time scheduling, and requirements for (4) fault tolerance, (5) tight clock synchronization, and (6) highly predictable message transfers on the network. Since none of these problems is trivial, they have been a subject of active research for the past decades, and the results are in abundance. For example, many researchers have tackled distributed scheduling, and there are many results for this NP-hard problem [3]. Clock synchronization techniques have been rigorously explored by Lamport [5], and further developed by Kopetz [4]. As for real-time communication, particularly in industrial applications, the fieldbuses have been proposed for cost-effective, predictable communication systems, and now become widely accepted standards for developing industrial real-time systems [8, 12].

For the timing constraint derivation problem, we have developed an end-to-end design methodology in [10, 11] in that the system-level timing requirements of a design are automatically mapped onto component-level timing constraints according to its functional structure and resource constraints. We believe that this design method aids designers in achieving the temporal composibility of distributed real-time system design, and thus helps manage its inherent complexity. Note that it is often extremely difficult to design a distributed real-time system in a completely composible fashion in the presence of timing constraints: temporal relationships induced by the timing constraints may introduce coupling between structurally irrelevant components. Thus, if a mapping from system-level constraints onto task-level constraints can be found, the composibility of the distributed real-time system design can be achieved.

On the other hand, the end-to-end design method was developed under a number of simplifying and generalizing assumptions which enable us to ignore interactions with other design problems such as task allocation, clock synchroniza-

*The work reported in this paper was supported in part by MOST under the National Research Laboratory (NRL) grant 2000-N-NL-01-C-136, by Automatic Control Research Center (ACRC), and by Automation and Systems Research Institute (ASRI).

tion, and network scheduling. For example, we assumed that systems possess a perfectly synchronized global time base with which timing constraints are enforced, that every communication message is transfered in a bounded amount of time, and that task scheduling policies are not fixed.

In this paper, we attempt to validate the end-to-end design method and assess its practical utility through a case study on a distributed numerical controller that is built on the FIP network. In doing so, we extend the original method to remove the abovementioned assumptions. First, we enhance the deadline decomposition algorithm given in [10] to increase the chances of producing schedulable timing constraints. Second, we refine the intermediate constraint derivation process to take into account the limitations of the FIP network scheduling. Finally, we incorporate clock synchronization into the timing constraint derivation process. Note that for this paper, we do not address the task allocation problem.

Figure 1 shows an overview of the extended design methodology. A system design specified with a task graph and its system-level timing requirements are given as input. Also given as an input is the task allocation which is assumed to be static. Then the task-level timing constraints are derived through well-defined steps such as intermediate constraint derivation, period assignment, and phase and deadline assignment. Our algorithm iteratively refines the task-level timing constraints until a schedulable solution is found.

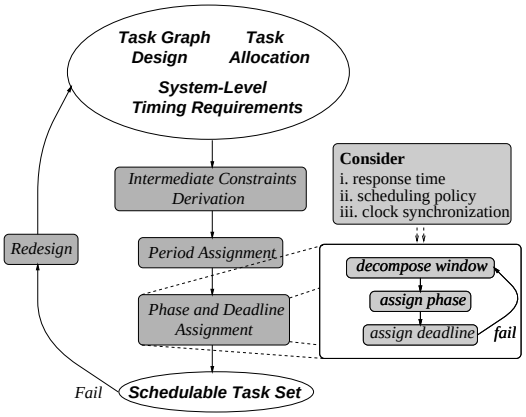

Figure 1: The structure of the end-to-end design methodology.

We have built a numerical control system on a distributed platform which consists of the FIP network and three Pentium PCs. Each PC runs the ARX real-time operating system that we have developed at Seoul National University [2]. We have generated a controlled series of dozens of controllers and performed extensive experiments. These experiments clearly show that the new deadline decomposition algorithm outperforms existing ones, and that the end-to-end design method can be effectively applied to the design of highly predictable industrial real-time systems, if it is properly customized for the given hardware and software platform.

2 System Model

In this section, we describe the system model and timing constraints: (1) per task specification, (2) task graph, and (3) system-level timing constraints.

2.1 Per Task Specification

Each periodic task τ_i is specified by a 5-tuple $\langle e_i, T_i, d_i, \phi_i, \mathcal{F}_P(\tau_i)\rangle$ where e_i represents the task's execution time, T_i its period, d_i its relative deadline, ϕ_i its phase, and $\mathcal{F}_P(\tau_i)$ denotes a mapping of a task τ_i to a processor to which τ_i is allocated. We assume that e_i and $\mathcal{F}_P(\cdot)$ are given as inputs according to a static mapping between tasks and processors. The other components (i.e., T_i, d_i, and ϕ_i) are unknown variables and are determined by the end-to-end approach. Similarly, a message stream m_i is modeled as a periodic task with $\langle e_i^m, T_i^m, d_i^m, \phi_i^m, \mathcal{F}_P(m_i)\rangle$ where e_i^m is the communication delay, T_i^m its period, d_i^m its relative deadline, ϕ_i^m its phase, and $\mathcal{F}_p(m_i)$ the network resource.

2.2 Task Graph

A directed graph $G(V, E)$ called a task graph is used to model the interaction among tasks in the system where

- $V = \mathcal{T}_{compute} \cup \mathcal{T}_{communicate}$ is a set of nodes where $\mathcal{T}_{compute}$ is the set of computation tasks, and $\mathcal{T}_{communicate}$ is the set of communication tasks.
- $E \subset (\mathcal{T}_{compute} \cup \mathcal{T}_{communicate}) \times (\mathcal{T}_{compute} \cup \mathcal{T}_{communicate})$ is a set of directed edges between nodes that denote producer/consumer relationships between tasks.

A computation task in a task graph reads data from its input ports which are denoted m_i, performs computation based on the input data, and finally writes the results into its output ports. On the other hand, a communication task delivers a message, which corresponds to reading the message from a sender and writing the message to a receiver after the communication delay.

In a task graph, a path from a sensor to an actuator is a sequence of producer/consumer pairs forming an end-to-end computation. We denote by $\gamma(S||A)$ a *task path* that takes an input from sensor S and produces an output to actuator A. When a task τ_i is on a path γ_k, we represent this by $\tau_i \in \gamma_k$. Similarly, we denote by $\Gamma(S_m, \ldots, S_n||A)$ a *transaction* that takes inputs from multiple sensors $S_m, \ldots, S_n$ and produces an output to a single actuator A.

Figure 2 shows the task graphs of the numerical control system that was used in our case study. It was modeled

as three task graphs which collectively possessed control tasks, sensors, and actuators, for feedback control. One task graph corresponds to a controller subsystem, and the others a monitoring subsystem. Tasks τ_1 through τ_7 are controller tasks which read position values from sensors and generate control commands to actuators; tasks τ_8 and τ_9 are monitor tasks.

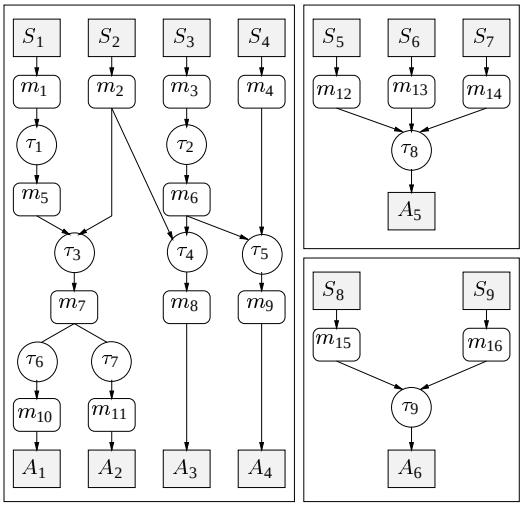

Figure 2: Task graphs.

2.3 System-Level Timing Constraints

There are two types of system-level timing constraints that are placed on transactions.

- *Maximum Transaction Period (MTP)*: This bounds the maximum activation period for an end-to-end computation. Notation $MTP(S_1, S_2||A)$ is used to denote the maximum period of transaction $\Gamma(S_1, S_2||A)$.
- *Maximum Transaction Delay (MTD)*: This bounds the maximum allowable end-to-end delay between the reading of a sensor and the writing to an actuator. Notation $MTD(S_1, S_2||A)$ is used to denote the maximum delay from sensors S_1 and S_2 to an actuator A.

Table 1 gives the system-level timing constraints for the example in Figure 2.

2.4 Problem Description and Solution Overview

Given a task graph design of a system with the task allocation and the system-level timing requirements, the problem is to derive task attributes for each task and message subject to chosen scheduling policies and the clock synchronization requirement. This problem is modeled as a constraint solving problem in which the system-level timing requirements are first expressed as a set of intermediate constraints on the task attributes. Then these intermediate constraints are solved in such a way that the results preserve the timing correctness: if the final task set (which is a solution of the constraints) is schedulable, then the original system-level requirements will be satisfied. The approach taken in our methodology is to decompose the constraint solving problem into a sequence of sub-problems. The motivation behind this approach is that specialized heuristics and performance metrics may be employed for each sub-problem. For this paper, this problem is decomposed into the *Period Assignment* and the *Phase/Deadline Assignment* sub-problems.

In section 3, we explain the original end-to-end approach with an improved deadline decomposition algorithm, and in section 4, we present FIP specific extensions to this approach.

Table 1: System-level timing constraints (μs)

Transaction	*MTP*	*MTD*
$\Gamma_1(S_1, S_2\|\|A_1)$	14400	19200
$\Gamma_2(S_1, S_2\|\|A_2)$	42000	45600
$\Gamma_3(S_2, S_3\|\|A_3)$	12000	18000
$\Gamma_4(S_3, S_4\|\|A_4)$	24000	30000
$\Gamma_5(S_5, S_6, S_7\|\|A_5)$	11000	12000
$\Gamma_6(S_8, S_9\|\|A_6)$	12000	14400

3 End-to-End Design Methodology

In this section we discuss the two ingredients of the end-to-end design methodology, namely, intermediate constraint derivation and period and phase/deadline assignment. Particularly, we present a new and improved deadline decomposition algorithm that is based on execution window decomposition.

3.1 Intermediate Constraint Derivation

3.1.1 Intermediate Constraints from the Producer/Consumer Model and MTP

The producer/consumer model forms a basic communication semantics in our transaction model. A producer/consumer pair inherently incurs blocking synchronization in that a consumer task must wait for a producer to generate the required data. For all producer/consumer pairs, we restrict a producer's period to be an integer multiple of its consumer's so that we can reduce a communication delay between, otherwise, arbitrarily-phased tasks. This relationship is referred to as being *harmonic* and is represented as $T_p|T_c$, where the operator "|" is interpreted as "exactly divides." Note that harmonic periods also simplify task scheduling.

Aside from the harmonicity constraint, the *MTP* requirement imposes an upper bound on the period for each task. When a task is shared by several transactions the minimum upper bound must be chosen to guarantee all the other *MTP* requirements.

3.1.2 Intermediate Constraints from Precedences and MTD

One of the system-level timing properties of interest is the end-to-end delay from a sensor reading to an actuator output. Since data flow through the tasks on the path from the sensor to the actuator and delay is incurred at each step, it is necessary to compute the intermediate delays. This flow of data is a producer/consumer model. Recall that under producer/consumer synchronization a precedence relation between the producer and consumer is required. In distributed real-time systems, an attractive way to enforce this precedence is through the use of task phase variables. Thus, for a given producer/consumer pair (τ_p, τ_c) and message task τ_p^m between them, the following intermediate constraints due to precedence constraints ($\tau_p \rightarrow \tau_p^m \rightarrow \tau_c$) are enforced.

$$\phi_p + d_p \leq \phi_p^m \quad \text{and} \quad \phi_p^m + d_p^m \leq \phi_c$$

The next intermediate constraint of the producer/consumer pair (τ_p, τ_c) has to do with an end-to-end delay from a sensor to an actuator. The maximum delay from the time τ_p reads data from its input ports to the time τ_c writes data to its output ports is when τ_p reads its input data just as it is invoked and τ_c finishes writing to the output ports at the end of its activation period. Therefore, the worst case delay becomes $\phi_c + d_c - \phi_p$. Consider a transaction $\Gamma(S||A) = (\tau_1 \rightarrow \tau_2 \rightarrow \ldots \rightarrow \tau_n)$. By extending the same logic to $\Gamma(S||A)$, the following end-to-end intermediate constraint must be satisfied.

$$\phi_n + d_n - \phi_1 \leq MTD(S||A)$$

3.2 Period Assignment

Once the intermediate constraints are derived, the next step is to solve them for task attributes. This process is further subdivided into two steps. The first is the period assignment step which is constrained by two types of constraints.

- *Range constraints:* These constraints ensure a minimum rate of task executions, and therefore, impose an upper-bound on periods. An implicit lower bound exists for a period variable, which arises from the task execution time.
- *Harmonicity constraints:* These constraints arise from synchronous communication between each producer/consumer task pair.

With these constraints, the objective chosen in this step is minimizing total utilization and balancing utilizations among processors. For this, we have developed a polynomial-time period assignment algorithm in [9] which works in two steps, as below.

Step 1: It first sorts tail tasks (which do not have any consumer tasks) in the ascending order of the upper bounds of their periods, and assigns them harmonic periods such that the period of the $(k+1)^{th}$ task in the sorted list is the largest integer multiple of that of the k^{th} task.

Step 2: It assigns each non-tail task the greatest common divisor (GCD) of the periods of its consumer tasks.

The GCD-based assignment of Step 2 guarantees that given the period assignment of tail tasks, non-tail tasks will have a period assignment that leads to the minimum total utilization. Thus, the problem ends up finding an optimal period assignment for only tail tasks. To this end, Step 1 assigns harmonic periods to tail tasks: this increases the greatest common divisor (GCD) of tail tasks possessing the same producers. We have proven in [9] that the utilizations of the solutions this algorithm generates are no greater than twice the optimal utilization. In practice, however, the resultant utilization is very close to the optimal one in most cases, since the proof is derived under the worst-case assumption. Finally, to balance utilizations among processors, we select one of derived solutions that minimizes $\max(U_i)$ for all processors $\mathcal{P}_i$ in the system where U_i is $\mathcal{P}_i$'s utilization.

3.3 Phase/Deadline Assignment

Once periods are determined, the next step is to assign phases and deadlines to tasks. This is made possible via execution window decomposition, whose example is given in Figure 3. In the example, an *MTD* constraint is placed on a task path $\gamma(S||A) = (\tau_1 \rightarrow \tau_2 \rightarrow \tau_3)$,

$$w_1 + w_2 + w_3 \leq D_1 \qquad \textit{(Eq 1)}$$

where D_1 is the allowable end-to-end delay for task path $\gamma(S||A)$. This constraint bounds the sum of the execution windows for tasks τ_1, τ_2, and τ_3. In other words, the end-to-end delay D_1 is decomposed into intermediate windows w_i, for $i = 1, 2, 3$. The solution to this problem should assign a sufficient time budget to each task for feasible scheduling while meeting the end-to-end delay imposed on the transaction. When there are more than one path in a transaction, each path is subject to the end-to-end delay constraint on the transaction.

Unfortunately, it is extremely difficult to determine task windows at this step optimizing task schedulability, since schedulability is dependent on decisions that are made at implementation stage. There are a few approaches that address this problem using processor utilization as a measure of schedulability [11, 10]. In them, tasks get relatively large windows, when they are hosted on a processor with large

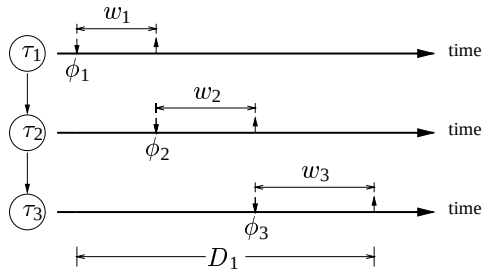

Figure 3: End-to-end delay decomposition.

utilization. Specifically, for task τ_i on task path γ_k with end-to-end delay D_k, task window w_i is assigned proportionally to the utilization of τ_i's host, as shown below. Here, $\mathcal{F}_U(\tau_i)$ denotes the utilization of processor $\mathcal{F}_P(\tau_i)$.

$$w_i = \frac{\mathcal{F}_U(\tau_i)}{\sum_{\tau_j \in \gamma_k} \mathcal{F}_U(\tau_j)} D_k \qquad \textit{(Eq 2)}$$

Though this approach appears sensible, it does not work well in practice for two reasons. First, processor utilization is an insufficient measure, where a task set fails to follow the assumptions made for rate monotonic scheduling [6]. Second, scheduling policies sometimes play a more important role than processor utilization, in determining schedulability. Thus, for distributed real-time system models, the following factors must be taken into consideration.

- *Task response time*: Response time is a direct and sufficient measure for schedulability. Note that a task is schedulable if its response time is no greater than its deadline.
- *Scheduling policy*: Since each scheduling policy uses a different scheduling strategy, the response time of a task varies with scheduling policies.

Based on these observations, our solution strategy relies on a heuristic called *fixed plus proportional* partitioning which is summarized below.

$$w_i = e_i + \Upsilon_i L_k \qquad \textit{(Eq 3)}$$

The fixed (constant) component e_i ensures the minimum execution interval for task τ_i, and the proportional one distributes *path slack* L_k according to ratio Υ_i where the path slack is defined below.

$$L_k \triangleq D_k - \sum_{\tau_l \in \gamma_k} e_l \qquad \textit{(Eq 4)}$$

Υ_i is a product of the above two factors: *approximate response* $\tilde{R}_i$ and *scheduling performance index* Ψ_i.

$$\Upsilon_i \triangleq \frac{\tilde{R}_i \Psi_i}{\sum_{\tau_j \in \gamma_k} \tilde{R}_j \Psi_j} \qquad \textit{(Eq 5)}$$

Note that the denominator is used to make the sum of Υ_i on task path γ_k 1.0.

$\tilde{R}_i$ approximates the response time R_i of τ_i where R_i can be computed as below under fixed-priority preemptive scheduling [1].

$$\begin{aligned} R_i &= e_i + I_i \\ I_i &= \sum_{\tau_j \in \mathcal{T}_{hp(\tau_i)}} \lceil \frac{R_i}{T_j} \rceil e_j \end{aligned} \qquad \textit{(Eq 6)}$$

Here, $\mathcal{T}$ is a set of tasks hosted on $\mathcal{F}_p(\tau_i)$, and $\mathcal{T}_{hp(\tau_i)}$, a subset of $\mathcal{T}$, contains all higher priority tasks than τ_i. I_i is the time interfered by tasks in $\mathcal{T}_{hp(\tau_i)}$. As can be seen from equation (6), we need to know $\mathcal{T}_{hp(\tau_i)}$ to compute R_i. However, we do not know it until all the deadlines of tasks in $\mathcal{T}$ are determined. To break this circular dependency, we make a worst-case assumption that $\mathcal{T}_{hp(\tau_i)} = \mathcal{T} - \{\tau_i\}$. Then, we rewrite equation (6) with $\tilde{R}_i$ and $\tilde{I}_i$.

$$\begin{aligned} \tilde{R}_i &= e_i + \tilde{I}_i \\ \tilde{I}_i &= \begin{cases} 0 & \text{if } e_i = 0 \\ \sum_{\tau_j \in \mathcal{T} - \{\tau_i\}} (\frac{\tilde{R}_i}{T_j} + 1) e_j & \text{otherwise} \end{cases} \end{aligned} \qquad \textit{(Eq 7)}$$

After simple arithmetic manipulation, we have the following equation on $\tilde{R}_i$ ($e_i \neq 0$).

$$\tilde{R}_i = \frac{\sum_{\tau_j \in \mathcal{T}} e_j}{1 + \frac{e_i}{T_i} - \mathcal{F}_U(\tau_i)} \qquad \textit{(Eq 8)}$$

Υ_i also depends on scheduling performance index Ψ_i. Ψ_i is a tunable parameter which captures the degree of effectiveness of distinct scheduling policies. In our approach, Ψ_i is chosen via iterative refinement, as will be described in Algorithm 4.1.

After the intermediate execution windows are determined, the phase and deadline of task τ_i are computed with the following rules.

$$\begin{aligned} \phi_i &= \begin{cases} 0 & \text{if } i = 1 \\ \phi_{i-1} + w_{i-1} & \text{otherwise} \end{cases} \\ d_i &= w_i \end{aligned} \qquad \textit{(Eq 9)}$$

The fixed plus proportional partitioning is applied task by task according to the following rules.

- For each task τ_i shared by multiple task paths, a *critical path* γ_k is identified such that γ_k has the smallest path slack among the task paths; then ϕ_i and d_i are computed with respect to critical path γ_k using equation (3).
- For each task τ_i belonging to a single task path γ_k, ϕ_i and d_i are computed with respect to γ_k using equation 3.

These rules give shared tasks precedence over non-shared ones. Since shared tasks are subject to tight constraints that come from their critical paths, we first derive the phases and deadlines for the shared tasks. We then eliminate the shared tasks in the task graph, and this results in a set of sub-graphs which are constrained by the phases and deadlines of the shared tasks. We again apply the fixed plus proportional partitioning for the non-shared tasks.

4 FIP-Specific Extensions

In this section, we customize the original end-to-end design approach by taking into account the clock synchronization and network scheduling of FIP. We then present a modified phase/deadline assignment algorithm.

4.1 Message Scheduling in FIP

The scheduling mechanism incorporated into the FIP is a simple and efficient one which resembles cyclic executive scheduling [7]. The FIP designates a bus arbiter which controls data exchanges between interconnected nodes. When a system is configured, the bus arbiter gets a bus arbitrating table which contains a cyclic schedule of periodic messages. At run time, it scans periodic variables in the table and initiates data exchanges. A major issue in the FIP bus scheduling is how to organize macro and elementary cycles. Figure 4 shows an example schedule for periodic variables A, B, C, and D.

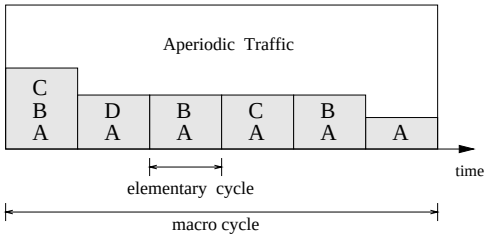

Figure 4: Macro and elementary cycles in a cyclic schedule.

The simplicity of the FIP message scheduling allows a practical solution for timely message transfers. Yet, it may also seriously compromise flexibility in scheduling, and this may result in undesirable message blocking. Consider an example in Figure 5 where two periodic messages m_a and m_b are scheduled. If they are scheduled in the same elementary cycle, as in (A), there must be a pause between m_a and m_b, due to phase constraint ϕ_b on m_b. However, such a pause is not realizable in the FIP scheduling, since the bus arbiter cannot enforce exact timing but message ordering within an elementary cycle. Thus, as in the cyclic schedule of (B), m_b must be scheduled in the subsequent elementary cycle. This implies that a message with a nonzero phase can be blocked for an elementary cycle in the worst case. We call this *phase blocking*.

In order to eliminate phase blocking, we restrict phase values to be an integer multiple of the elementary cycle. This

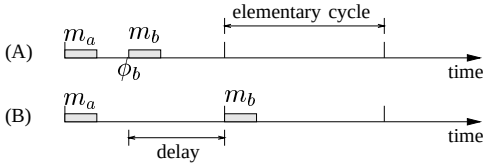

Figure 5: Additional blocking in the FIP scheduling: (A) an ideal schedule, and (B) a real schedule.

enables us to maintain the simplicity of the FIP scheduling without introducing additional blocking. To reflect this idea, we modify the phase/deadline assignment. Given a task path γ_k ($\tau_1 \rightarrow \tau_2 \rightarrow \ldots \tau_n$), and elementary cycle size E, the fixed plus proportional partitioning is modified as below.

$$L_k^* = D_k - \sum_{\tau_l \in \gamma_k} \lceil \frac{e_l}{E} \rceil E$$

$$w_i = \lceil \frac{e_l}{E} \rceil E + \lfloor \frac{\Upsilon_i L_k^*}{E} \rfloor E \qquad \textit{(Eq 10)}$$

4.2 Clock Synchronization in FIP

It is desirable to keep a global clock for time-based design approaches such as ours, since a global time base can significantly simplify synchronizations between producer/consumer pairs. The FIP has a potential for offering a synchronized global clock, since every communicating node on the FIP network is connected with a single bus, and message transfers are controlled by a single bus arbiter. However, the FIP itself does not specify clock synchronization at the application layer. We thus present a solution to clock synchronization which makes use of a periodic sync variable message and interrupt driven communication between a host and an FIP board.

Generally, clock synchronization errors are due to *clock drifts* and *faulty clocks* [5, 4]. Every distributed physical clock has a non-zero drift rate, and may be faulty from time to time. Clock synchronization errors are also classified into *internal* and *external*. The internal synchronization error denoted Δ_{int} is the maximum difference between two local clocks in the system. The external one denoted Δ_{ext} is the maximum difference between a local clock and the external reference clock. Most of the existing solutions to clock synchronization attempt to periodically resynchronize local clocks to bound synchronization errors by a known constant. We consider clock resynchronization in an FIP-based system.

In an FIP-based system, hosts are equipped with FIP boards which are connected through a single bus. When a host starts up, its FIP board downloads the network-related information and initializes its resources. At runtime, message exchanges between the FIP boards are synchronized with the clock of the bus arbiter (bus clock). On the other

hand, application programs are not synchronized with the bus clock, for the lack of clock synchronization at the application layer. Thus, for an FIP-based system with n hosts, there exist $n+1$ logical clocks: n for the hosts and 1 for the bus. We use the bus clock as an internal reference.

In order to enable application programs to participate in synchronous message communication, we use an extra *sync* variable and the interrupt mechanism between hosts and the FIP boards. The bus arbiter sends out a sync variable at the beginning of every elementary cycle, and the FIP boards generate an interrupt upon the receipt of the variable. Then the application programs adjust their clocks according to the bus clock. Note that for FIP-based systems, we need not consider faulty clocks, since only the bus arbiter can affect other hosts; when the bus arbiter fails, another host must be elected anyway.

In the absence of faulty clocks, the synchronization error can be stated as the sum of *reading error* ϵ_r and *drift error* ϵ_d [4, 5], as below. The reading error is the maximum difference between clock reading delays on any two hosts. Given the maximum drift rate $\hat{\rho}$ of clocks and the resynchronization period T_r, the drift error is the product of $\hat{\rho}$ and T_r.

$$\Delta_{int} \quad = \quad \epsilon_r + \hat{\rho}\, T_r \qquad \textit{(Eq 11)}$$

To compensate for internal synchronization errors, we must tighten deadlines as below.

$$\begin{aligned} \phi_i &= \phi_{i-1} + w_{i-1} \\ d_i &= w_i - \Delta_{int} \end{aligned} \qquad \textit{(Eq 12)}$$

Similarly, for external synchronization errors, we must tighten MTP and MTD by Δ_{ext}. For the purpose of presentation, we assume Δ_{ext} to be zero.

5 A Case Study

The goals of this case study are to (1) demonstrate the utility of the extended approach, (2) validate its correctness from the standpoint of timing correctness, and (3) assess the effectiveness of the chosen heuristics.

5.1 Hardware/Software Platform

The distributed platform consisted of three Pentium PCs and three FullFIP2 boards which were installed on these PCs. These FIP boards were connected through shielded twisted pair wire. Each PC ran the ARX real-time operating system that was developed at Seoul National University [2]. The ARX provided lightweight user-level threads and periodic real-time tasks were implemented with them. The FIP software package (FIP Toolbox v.3.2) was ported on the ARX. Figure 6 pictorially depicts the platform and task allocation. PC $\mathcal{P}_1$ hosted four tasks $\langle \tau_1, \tau_3, \tau_6, \tau_7 \rangle$, and PC $\mathcal{P}_2$ five tasks $\langle \tau_2, \tau_4, \tau_5, \tau_8, \tau_9 \rangle$; PC $\mathcal{P}_3$ interfaced nine sensors and four actuators with the system, also working as a bus arbiter.

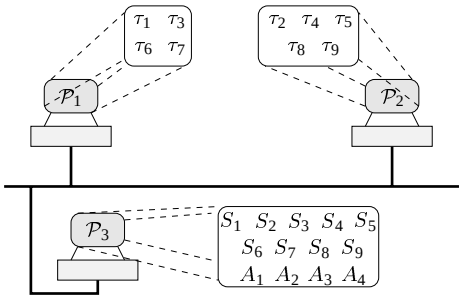

Figure 6: Task allocation on the distributed platform.

Table 2 gives the execution times of the tasks in the system. The execution times include time to read from or write into the data buffer in the data link layer. They were measured on Pentium processors using a built-in event counter. The execution times of message stream tasks were calculated with the transmission speed being 1Mb/s, according to the FIP specification. Since internal message tasks did not go through the FIP network, their execution times were zero. As a result, $e_i^m = 0$ for $i = 5, 6, 7$; $e_i^m = 250\mu s$, otherwise.

Table 2: Task execution times (μs)

Task	τ_1	τ_2	τ_3	τ_4	τ_5
Exec. Time	800	600	500	1100	1400
Task	τ_6	τ_7	τ_8	τ_9	·
Exec. Time	1000	1200	700	500	

5.2 Derivation of Task Attributes

As described earlier, task attributes are derived from the given end-to-end timing constraints via a set of intermediate constraints. For simplicity, the constraint derivation process is not discussed in this section.

Period assignment is performed using the algorithm described in [9], and the result is shown in Table 3. Finally, phase/deadline assignment is performed. First, eight shared tasks and their critical paths are identified in Table 4. This yields an instance of the execution window decomposition problem possessing six inequalities, as shown below. Note that the same critical paths are selected for τ_3 and m_7 and for m_6 and τ_4.

$$\begin{gathered} d_2^m + d_3 + d_7^m + d_6 + d_{10}^m \leq 19200 \\ d_1^m + d_1 + d_5^m + d_3 + d_7^m + d_6 + d_{10}^m \leq 19200 \\ d_3^m + d_2 + d_6^m + d_4 + d_8^m \leq 18000 \\ d_3^m + d_2 + d_6^m + d_5 + d_9^m \leq 30000 \\ d_{12}^m + d_8 \leq 12000 \\ d_{15}^m + d_9 \leq 14400 \end{gathered}$$

Table 3: Task periods (μs)

Task	τ_1	τ_2	τ_3	τ_4	τ_5
Period	12000	12000	12000	12000	24000
Task	τ_6	τ_7	τ_8	τ_9	m_1
Period	12000	48000	12000	12000	12000
Task	m_2	m_3	m_4	m_5	m_6
Period	12000	12000	24000	12000	12000
Task	m_7	m_8	m_9	m_{10}	m_{11}
Period	12000	12000	24000	12000	48000
Task	m_{12}	m_{13}	m_{14}	m_{15}	m_{16}
Period	12000	12000	12000	12000	12000

Table 4: Shared tasks and critical paths

Task	Critical Path
m_2	$m_2 \rightarrow \tau_3 \rightarrow m_7 \rightarrow \tau_6 \rightarrow m_{10}$
τ_3	$m_1 \rightarrow \tau_1 \rightarrow m_5 \rightarrow \tau_3 \rightarrow m_7 \rightarrow \tau_6 \rightarrow m_{10}$
m_7	$m_1 \rightarrow \tau_1 \rightarrow m_5 \rightarrow \tau_3 \rightarrow m_7 \rightarrow \tau_6 \rightarrow m_{10}$
m_6	$m_3 \rightarrow \tau_2 \rightarrow m_6 \rightarrow \tau_4 \rightarrow m_8$
τ_4	$m_3 \rightarrow \tau_2 \rightarrow m_6 \rightarrow \tau_4 \rightarrow m_8$
τ_5	$m_3 \rightarrow \tau_2 \rightarrow m_6 \rightarrow \tau_5 \rightarrow m_9$
τ_8	$m_{12}\,(m_{13}, m_{14}) \rightarrow \tau_8$
τ_9	$m_{15}\,(m_{16}) \rightarrow \tau_9$

Each of the above constraints is solved by fixed plus proportional partitioning. The final results are summarized in Table 5; note that internal messages have zero deadlines, since their execution times are zero.

Table 5: Phases and deadlines (μs)

Task	τ_1	τ_2	τ_3	τ_4	τ_5
Phase	4000	4000	6000	6000	14000
Deadline	1835	3835	1835	1835	5835
Task	τ_6	τ_7	τ_8	τ_9	m_1
Phase	8000	8000	6000	8000	0
Deadline	1835	11835	3835	5835	3835
Task	m_2	m_3	m_4	m_5	m_6
Phase	0	0	0	6000	8000
Deadline	3835	3835	13835	0	0
Task	m_7	m_8	m_9	m_{10}	m_{11}
Phase	8000	10000	20000	10000	20000
Deadline	0	3835	7835	3835	23835
Task	m_{12}	m_{13}	m_{14}	m_{15}	m_{16}
Phase	0	0	0	0	0
Deadline	5835	5835	5835	7835	7835

6 Conclusion

In this paper, we described a case study of applying our end-to-end method to the design of an FIP-based distributed real-time system. This study revealed that the original design method had several limitations due to simplifying assumptions it possessed. Thus, we extended it by taking into account practical issues such as clock synchronization and network scheduling. We also improved its deadline decomposition algorithm.

The design approach can be improved further. For example, we can consider task allocation and timing constraint derivation in an integrated approach so that valuable feedback can be given for task allocation when a feasible solution is not found. We are currently investigating such improvement.

References

[1] N. Audsley, A. Burns, M. Richardson, and A. Wellings. Hard real-time scheduling: The deadline-monotonic approach. In *Proceedings of IEEE Workshop on Real-Time Operating Systems and Software*, pages 133–137, May 1991.

[2] S. Hong, Y. Seo, and J. Park. ARX/ULTRA: A new real-time kernel architecture for supporting user-level threads. Technical Report SNU-EE-TR-1997-3, School of Electrical Engineering, Seoul National University, Korea, August 1997.

[3] C. J. Hou and K. Shin. Allocation of periodic task modules with precedence and deadline constraints in distributed real-time systems. *Proceedings of IEEE Real-Time Systems Symposium*, 1992.

[4] H. Kopetz and W. Ochsenreiter. Clock synchronization in distributed real-time systems. *IEEE Transactions on Computers*, C-6(8):933–940, 1987.

[5] L. Lamport and P. M. Melliar-Smith. Synchronizing clocks in the presence of faults. *Journal of the Association for Computing Machinery*, 32(1):52–78, 1985.

[6] 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.

[7] C. Locke. Software architecture for hard real-time applications: Cyclic executives vs. fixed priority executives. *The Journal of Real-Time Systems*, 4(1):37–53, March 1992.

[8] P. Pleinevaux and J. D. Decotignie. Time critical communication networks: Field buses. *IEEE Network Magazine*, May 1988.

[9] M. Ryu and S. Hong. A period assignment algorithm for embedded digital controllers. Technical Report SNU-EE-TR-96-1, School of Electrical Engineering, Seoul National University, Korea, October 1996.

[10] M. Ryu and S. Hong. End-to-end design of distributed real-time systems. *Workshop on Distributed Computer Control Systems*, pages 22–27, July 1997.

[11] M. Saksena and S. Hong. Resource conscious design of real-time systems: An end-to-end approach. *IEEE International Conference of Engineering Complex Computer Systems*, pages 306–314, October 1996.

[12] WorldFIP. *General Purpose Field Communication System, prEN 50170*. WorldFIP, 1995.