CASPER: Concurrent Hardware-Software Co-Synthesis of Hard Real-Time Aperiodic and Periodic Specifications of Embedded System Architectures*

Bharat P. Dave[1] and Niraj K. Jha

Department of Electrical Engineering
Princeton University, Princeton, NJ 08544

Abstract

Hardware-software co-synthesis of an embedded system requires mapping of its specifications into hardware and software modules such that its real-time and other constraints are met. Embedded system specifications are generally represented by acyclic task graphs. Many embedded system applications are characterized by aperiodic as well as periodic task graphs. Aperiodic task graphs can arrive for execution at any time and their resource requirements vary depending on how their constituent tasks and edges are allocated. Traditional approaches based on a fixed architecture coupled with slack stealing and/or on-line determination of how to serve aperiodic task graphs are not suitable for embedded systems with hard real-time constraints, since they cannot guarantee that such constraints would always be met. In this paper, we address the problem of concurrent co-synthesis of aperiodic and periodic specifications of embedded systems. We estimate the resource requirements of aperiodic task graphs and allocate execution slots on processing elements and communication links for executing them. Our approach guarantees that the deadlines of both aperiodic and periodic task graphs are always met. We have observed that simultaneous consideration of aperiodic task graphs while performing co-synthesis of periodic task graphs is vital for achieving superior results compared to the traditional slack stealing and dynamic scheduling approaches. To the best of our knowledge, this is the first co-synthesis algorithm which provides simultaneous support of periodic and aperiodic task graphs with hard real-time constraints. Application of the proposed algorithm to several examples from real-life telecom transport systems shows that up to 28% and 34% system cost savings are possible over co-synthesis algorithms which employ slack stealing and rate-monotonic scheduling, respectively.

1 Introduction

Architecture definition of an embedded system requires simultaneous synthesis of the hardware and software architectures which is usually referred to as hardware-software co-synthesis. Finding an optimal hardware-software architecture entails selection of processors, application-specific integrated circuits (ASICs) and communication links such that the cost of the architecture is minimum and all real-time constraints are met. Hardware-software co-synthesis involves various steps such as allocation, scheduling and performance estimation. The allocation step determines the mapping of tasks to processing elements (PEs) and inter-task communications to communication links. The scheduling step determines the sequencing of tasks mapped to a PE and sequencing of communications on a link. The performance estimation step estimates the finish time of each task and determines the overall quality of the architecture in terms of its dollar cost, ability to meet its real-time constraints, power consumption and fault tolerance *etc*. Both allocation and scheduling are known to be NP-complete [1]. Therefore, optimal co-synthesis is computationally a very hard problem.

* Acknowledgments: This work was supported in part by Bell Laboratories of Lucent Technologies under the Doctoral Support Program and in part by NSF under Grant No. MIP-9423574.

[1] also at Bell Laboratories, Lucent Technologies, Holmdel, NJ 07733.

Many embedded systems are characterized by both aperiodic and periodic tasks. Examples of such systems are: flight control systems, telecom systems, command and control systems, process control systems, automobile control systems, space shuttle avionics systems, defense control systems, *etc*. Periodic tasks arrive at regular intervals. Aperiodic tasks have random arrival times. Periodic task graphs generally have hard real-time constraints, whereas aperiodic task graphs can have either hard or soft real-time constraints. Many researchers have addressed co-synthesis of periodic task graphs. Also, there exists a large amount of literature on scheduling of aperiodic tasks for a given architecture which either minimizes the probability of failure to complete an aperiodic task by its hard deadline or minimizes its response time. In this paper, we study the problem of concurrent co-synthesis of aperiodic and periodic task graphs with hard real-time constraints. The problem of co-synthesis of aperiodic task graphs is a difficult one since such task graphs arrive for execution at any time and their resource requirements vary depending on how each constituent task and edge is allocated. To solve this problem, we estimate the size of execution slots and allocate them on PEs and links of the architecture to which constituent tasks and edges are allocated such that the deadlines are always met. We have observed that it is important to simultaneously consider aperiodic task graphs while performing co-synthesis of periodic task graphs to obtain an efficient architecture. The proposed techniques have been incorporated into our existing co-synthesis system, COSYN [2], and the resulting system is called CASPER (Co-synthesis of Aperiodic SPecification of Embedded system aRchitectures). We are not aware of any other scheduling (sub-problem of co-synthesis) or co-synthesis algorithm which guarantees that the deadlines of aperiodic task graphs with hard real-time constraints will always be met. We also establish the efficacy of our deadline-based scheduling technique with respect to two traditional techniques, slack stealing and rate monotonic scheduling (RMS), via experimental results.

2 Review of Related Work and Contributions

In this section, we review related previous work and describe our contributions.

2.1 Related work

Researchers have primarily focused their interest in the last several years on hardware-software partitioning, a major sub-problem in co-synthesis [3,4] where target embedded systems have one-CPU-one-ASIC architectures. Co-design frameworks for co-specification and co-simulation have been described in [5,6] where hardware/software partitioning is performed manually. These systems provide an integrated environment to manage both hardware and software in co-design projects. In the area of distributed system co-synthesis, the target architecture can employ multiple processors, ASICs, and field-programmable gate arrays (FPGAs). Two distinct approaches have been used to solve the distributed system co-synthesis problem: optimal [7,8] and heuristic [2,9,10]. Optimal approaches are applicable to only small co-synthesis problem instances (10 or so tasks).

None of the above co-synthesis algorithms support co-synthesis of aperiodic task graphs with hard real-time constraints which are found in many embedded systems.

There is a vast amount of literature in the area of scheduling of soft and hard aperiodic tasks [11]-[16] for a given architecture. A survey of scheduling techniques is provided in [11]. These techniques address only scheduling, and not co-synthesis. There are two possible approaches for scheduling of aperiodic tasks: 1) static scheduling where the schedule is defined *a priori*, and 2) dynamic (also referred to as "on-line") scheduling where the decision regarding execution of aperiodic tasks is made on-line. Static scheduling is generally used for periodic task graphs. In case of aperiodic tasks, though static scheduling requires some upfront knowledge of the tasks, it has less computational overhead. Aperiodic task graphs can be soft or hard. Soft aperiodic task graphs do not have fixed deadlines. Algorithms proposed for scheduling aperiodic tasks in [12]-[16] are based on the dynamic scheduling paradigm. These approaches either minimize the probability of not meeting the deadline during allocation of tasks on a given architecture or minimize the response times. Although a dynamic approach does not require prior knowledge of task characteristics, it suffers from the following inherent disadvantages: 1) it incurs a computational overhead in determining the most suitable PE to allocate an aperiodic task to, such that the aperiodic task deadline can be met, 2) it incurs an additional delay in transferring the aperiodic task to another PE in the event a deadline cannot be shown to be met for the aperiodic task on the PE it first arrived at, and 3) it cannot give a guarantee that deadlines will always be met. In [12,13], techniques are presented to handle dynamic scheduling of soft and hard aperiodic tasks for uniprocessor systems based on the concept of slack stealing from the existing schedule of periodic tasks. Their limitations are: 1) they ignore precedence among tasks, *i.e.* the inter-task communications, and 2) they cannot handle simultaneous scheduling of aperiodic and periodic tasks. In [14], dynamic scheduling of aperiodic tasks is considered for homogeneous multiprocessor systems. However, these techniques too do not take inter-task communication into consideration. In [15], dynamic scheduling of aperiodic task graphs with precedence constraints is considered, however, inter-task communication scheduling is ignored and the target architecture is restricted to a set of homogenous processors. In [16], deadline assignment for tasks of an aperiodic task graph is considered for dynamic scheduling. Both static and dynamic approaches can employ either preemptive or non-preemptive scheduling.

To the best of our knowledge, the problem of scheduling hard real-time aperiodic task graphs without the above-mentioned restrictive assumptions has not been considered for distributed heterogeneous systems.

2.2 Contributions of this paper

We propose a co-synthesis algorithm, called CASPER, employing a static scheduling method for both hard real-time aperiodic and periodic task graphs without the restrictive assumptions made by previous co-synthesis and scheduling techniques. Co-synthesis of aperiodic and periodic task graphs is simultaneously performed. Our scheduling technique employs a combination of preemptive and non-preemptive scheduling approaches to provide efficient schedules. Our algorithm guarantees that deadlines of hard real-time aperiodic and periodic task graphs are always met. It allows multiple types and forms of PEs and communication links, and supports both concurrent and sequential modes of communication and computation. It employs the concept of *association array* [2] to tackle the problem of multi-rate tasks. It supports task graphs where different tasks have different deadlines. It also supports pipelining of task graphs. The accuracy of its finish-time estimation step is enhanced by employing a deadline-based scheduling technique. Experimental results establish its efficacy over the traditional slack stealing [13] and RMS [17] based approaches.

3 The Co-Synthesis Framework

In this section, we describe the resource library, execution model, task graph parameters, and scheduling techniques which form the co-synthesis framework.

3.1 The resource library

Embedded system specifications are mapped to elements of a *resource library*, which consists of a PE library and a link library.

The PE library consists of various types of FPGAs, ASICs, and general-purpose processors. Each FPGA is characterized by: 1) the number of gates/flip-flops/programmable functional units (PFUs), 2) the boot memory requirement, 3) the number of pins, *etc*. Each ASIC is characterized by: 1) the number of gates, and 2) the number of pins. Each general-purpose processor is characterized by: 1) the memory hierarchy information, 2) communication processor/port characteristics, 3) the context switch time, *etc*.

The link library consists of various types of links such as point-to-point, bus, local area network. Each link is characterized by: 1) the maximum number of ports it can support, 2) an access time vector which indicates link access times for different number of ports on the link, 3) the number of information bytes per packet, 4) packet transmission time, *etc*.

3.2 The execution model

Each application-specific function executed by an embedded system is made up of several sequential and/or concurrent jobs. Each job is made up of several tasks. Tasks are atomic units performed by embedded systems. A task contains both data and control flow information. The embedded system functionality is usually described through a set of acyclic task graphs. Nodes of a task graph represent tasks. Tasks communicate data to each other, indicated by a directed edge. Task graphs can be periodic or aperiodic as shown in Figure 1. Each periodic task graph has an earliest start time (*EST*), period, and deadline (*dl*). Each task of a periodic task graph inherits the task graph's period. Each task in a periodic task graph can have a different deadline. Hard aperiodic task graphs have a specified deadline which must be met. Aperiodic task graphs are characterized by a parameter, Υ, denoting the minimum time interval between two consecutive instances of an aperiodic task graph. An aperiodic task graph may start at any time.

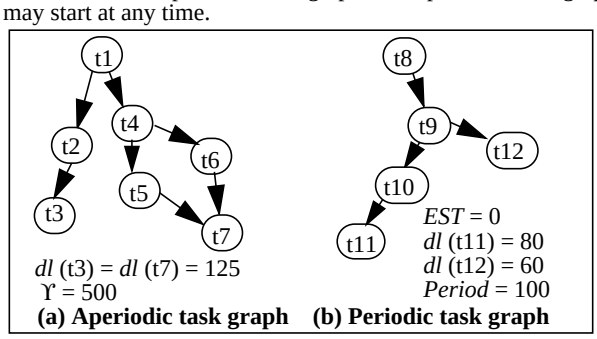

Figure 1. Aperiodic and periodic task graphs

Parameters used to characterize task graphs are described next. Each task is characterized by:

1. *Execution time vector*: This indicates the worst-case execution time of a task on the PEs in the PE library.
2. *Preference vector*: This indicates preferential mapping of a task on various PEs (such PEs may have special resources for the task).
3. *Exclusion vector*: This specifies which pairs of tasks cannot co-exist on the same PE (such pairs may create processing bottlenecks).
4. *Memory vector*: This indicates the different types of storage requirements for the task: program storage, data storage and stack storage.

A *cluster* of tasks is a group of tasks which is always allocated to the same PE. Clustering of tasks in a task graph reduces the communication times and significantly speeds up the co-synthesis process. Each cluster is characterized by the preference and exclusion vectors of its constituent tasks.

Each edge in the task graphs is characterized by:

1. The number of information bytes to be transferred.
2. *Communication vector*: This indicates the communication time for that edge on various links from the link library. It is computed based on link characteristics.

The communication vector for each edge is computed *a priori*. At the beginning of co-synthesis, since the actual number of ports on the links is not known, we use an average number of ports (specified beforehand) to determine the communication vector. This vector is recomputed after each allocation, considering the actual number of ports on the link.

In order to provide flexibility for the communication mechanism, we support two modes of communication: 1) sequential: where communication and computation cannot go on simultaneously and 2) concurrent: where communication and computation can go on simultaneously if supported by the associated communication link and PEs.

3.3 Scheduling

We use a static scheduler which employs a combination of preemptive and non-preemptive scheduling to derive efficient schedules. Tasks and edges are scheduled based on deadline-based priority levels (see Section 5.4). The schedule for real-time periodic and aperiodic task graphs is defined during architecture synthesis.

4 Co-Synthesis of Aperiodic Task Graphs

In this section, we discuss the problem of co-synthesis of hard real-time aperiodic task graphs, associated challenges, and techniques to address those challenges.

4.1 Problem description and challenges

The co-synthesis problem of periodic task graphs has been addressed in the literature before. However, an embedded system architecture must be capable of executing periodic and aperiodic task graphs concurrently such that the real-time constraints of all task graphs are met. Co-synthesis of aperiodic task graphs offers the following additional challenge:

- Aperiodic task graphs can arrive at the embedded system for execution at any time. Therefore, the architecture must have sufficient resources available at the required time to meet the deadline. Therefore, the resource requirements of such task graphs must be considered during architecture synthesis.

We formulate the above problem as an execution slot allocation problem. Execution slots are allocated to aperiodic task graphs similar to periodic task graphs on the architecture being synthesized such that their deadlines can always be met. There are two possible approaches for execution slot allocation: 1) determine the architecture based on periodic task graphs, follow up with execution slot allocation for aperiodic task graphs on the given architecture, and upgrade the architecture until all constraints of both periodic and aperiodic task graphs are met, or 2) determine the architecture by simultaneously considering periodic and aperiodic task graphs. We use the latter approach since simultaneous consideration of periodic and aperiodic task graphs results in very efficient architectures. This is demonstrated by experimental results. Another challenge is as follows.

- Aperiodic task graphs can have more than one task and communication edge. Tasks (edges) can potentially be mapped to a variety of PEs (links) since the architecture and allocation are not known beforehand. Therefore, one cannot exactly determine the length of the execution slot required from the start to finish of an aperiodic task graph. For example, as shown in the aperiodic task graph of Figure 1(a), there are several paths from the source node (*t1*) to the sink nodes (*t3*, *t7*). The length of each path (in terms of the execution and communication time) varies depending on the mapping of constituent tasks and edges, since there are numerous allocation possibilities for each task and edge.

We view the above problem as an execution slot size estimation problem. Next, we describe the techniques for solving the above two problems.

4.2 Execution slot size estimation

Allocation of periodic and aperiodic tasks is done simultaneously during the inner loop of the co-synthesis algorithm (see Section 5.3). We execute the aperiodic tasks at the next available execution slot. The hyperperiod of the system is computed as the least common multiple (LCM) of the periods of the various periodic task graphs in the specification. According to traditional real-time computing theory, a set of periodic task graphs has a feasible schedule if and only if it is schedulable in the hyperperiod [18]. We position execution time slots for aperiodic task graphs throughout the hyperperiod, Γ, such that the real-time constraints of both periodic and aperiodic task graphs are met irrespective of when the aperiodic task graph arrives for execution. Such a task graph can have one or more tasks. Since the architecture is not known *a priori*, the length, μ, of the execution time slot needs to be determined upfront in order to properly position these slots throughout the hyperperiod. We allow the user to specify μ based on his/her experience from existing designs or system specifications. If μ is not specified *a priori*, we use the following procedure to determine its value.

Let an aperiodic task graph T_j have m tasks, deadline dl_j (relative to *EST* of the task graph), minimum inter-instance time interval Υ_j, and let there be n PEs in the resource library. π_{is} represents the execution time of task i on PE s. We form clusters of tasks in T_j (using the method given in Section 5.2) and set the communication times of all intra-cluster communication edges to zero (this is based on the traditional assumption made in distributed computing that intra-PE communication takes zero time). We obtain the length of the longest path, $\Im$, in the clustered task graph using the maximum execution and communication times (from the corresponding execution/communication time vectors) for the associated tasks and edges, respectively. If the value of $\Im$ is greater than dl_j, we set its value equal to the length of the longest path which is less than or equal to dl_j. Next, we determine Θ and μ as follows (if task i is not allocatable to PE k based on an indication in the preference vector, then π_{ik} is set equal to zero to derive Θ).

$$S_k = \sum_{i=1}^{m} \pi_{ik} \qquad \text{(EQ 1)}$$

$$S_k = 0 \text{ if } S_k > dl_j \text{ or } \pi_{ik} = 0 \qquad \text{(EQ 2)}$$

$$\Theta = max(S_1, S_2, S_3, ..., S_n) \qquad \text{(EQ 3)}$$

$$\mu = max[\Im, \Theta] \qquad \text{(EQ 4)}$$

S_k represents the total time taken to execute task graph T_j on PE k assuming all tasks in T_j are allocated to PE k. If $S_k > dl_j$, then PE k cannot be chosen for allocating tasks from T_j since the deadline cannot be met. For this case, S_k is made zero so that PE k does not play a role in computing Θ and μ. If even one task of T_j cannot be allocated to PE k (based on the preference vector) then again PE k cannot be considered further. Thus, S_k is made zero for this case too. Θ represents the execution time of T_j on the PE on which it takes the most time to execute, while still ensuring that the deadline of T_j is met (usually such a PE would be the cheapest among the feasible PEs). $\Im$ represents the schedule length of T_j when not all of its tasks are allocated to the same PE. Note that different task clusters in T_j could potentially get allocated to different PEs and such PEs would be connected with various links. However, the schedule length for T_j cannot be allowed to exceed dl_j. Based on the above discussion, μ can be seen to be a large enough time interval to allow the co-synthesis algorithm to find a feasible single-PE or distributed architecture for T_j so that its deadline is guaranteed to be met. μ is used to determine the *EST* of each aperiodic task graph instance, as shown in the next section.

4.3 Execution slot allocation

In this section, we show how time slots of length μ can be distributed in the hyperperiod to tackle the aperiodic task graph no matter when it arrives. For the case $\mu \neq dl_j$, the minimum number of time slots in the hyperperiod required to tackle aperiodic task graph T_j with deadline dl_j is equal to $\phi = \lceil \Gamma \div (dl_j - \mu) \rceil$. When $\mu = dl_j$, all the time slots on the PE(s) the aperiodic task graph is allotted to should be available for it since the processing of such a task graph needs to start immediately. We assume the minimum inter-instance time interval, Υ_j, of T_j is greater than or equal to dl_j for the time being for simplicity of exposition. When $\Upsilon_j < dl_j$, we need to employ the concept of task graph pipelining (this is explained in Section 5.1). The allocated time slot has the form $\{y, z\}$, where y and z indicate its start- and finish-times, respectively. If the time slot is not available at the desired instant, we may need more execution time slots than ϕ. We position the first slot, assuming $EST = 0$, at $\{dl_j - \mu, dl_j\}$. Then, successive slots are positioned at $\{i(dl_j - \mu), i(dl_j) - (i-1)\mu\}$ throughout the hyperperiod Γ, where $i = 2, 3, ..., \phi$. Consider the required slot $\{r, s\}$. If $r \geq \Gamma$ and $s > \Gamma$, then we allocate a time slot at $\{r\text{-}\Gamma, s\text{-}\Gamma\}$ at the beginning of the hyperperiod. However, if $r < \Gamma$ and $s > \Gamma$, then allocated slots are $\{r, \Gamma\}$ and $\{0, s - \Gamma\}$. If the execution time slot is not available at the desired instant, say $\{w, z\}$, but is available earlier at $\{p, q\}$, then we allocate the execution time slot at $\{p, q\}$, the next slot at $\{p + dl_j - \mu, p + dl_j\}$, then at $\{p + 2(dl_j - \mu), p + 2dl_j - \mu\}$, and so on. If two slots overlap in time, they are merged into a larger slot.

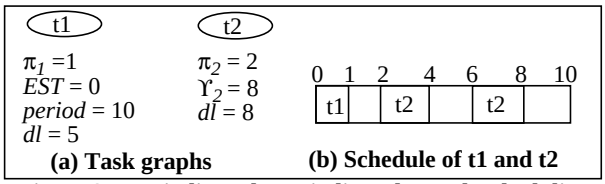

Figure 2. Periodic and aperiodic task graph scheduling

Consider the task graphs in Figure 2(a), where *t1* is periodic and *t2* is aperiodic. For simplicity, assume that there is only one task in each graph. π_1 and π_2 are the corresponding execution times on the sole PE in the resource library. Assume that both *t1* and *t2* are allocated to the same PE. The hyperperiod is 10. Since the aperiodic task graph has only one task, μ is equal to its worst-case execution time, which is equal to 2. The deadline of *t2* is equal to 8. Therefore, the number of execution time slots required by *t2* in the hyperperiod is $\lceil 10 \div (8-2) \rceil = 2$. The first execution time slot is required at {6, 8}. The second execution time slot is required at {12, 14}, which exceeds the hyperperiod. Thus, this slot is converted to {12–10, 14–10} = {2, 4}. Since this slot is available, it is allocated in the hyperperiod, as shown in Figure 2(b). Allocation of these two slots in the hyperperiod for *t2* guarantees that the deadline of *t2* is always met, irrespective of its arrival time, as long as two successive instances of *t2* are separated by Υ_2. If *t2* arrives before or at instant 2, it will be served by slot {2, 4}. If it arrives after instant 2 and before or at instant 6, it will be served by slot {6, 8}. Similarly, if it arrives after instant 6 and before or at instant 12, it will be served by the first slot of the next hyperperiod, and so on. Consider another example where $\pi = 4$, $dl = 10$, and $\Gamma = 15$. In this case, three slots are required as follows: {6, 10}, {12, 16}={12, 15}{0, 1}, and {18, 22} = {18-15, 22-15} = {3, 7}. Since slots {6, 10} and {3, 7} overlap, they are merged into one slot as {3, 10}.

Next, consider the more complex example shown in Figure 3(a). The specification consists of an aperiodic task graph *T1* and a periodic task graph *T2*. Suppose that the PE library consists of two PEs and the link library consists of a single link. The execution (communication) times of the different tasks (edges) on members of the PE (link) library are also shown in Figure 3(a). Since there is only one periodic task graph, its period is equal to the hyperperiod. Thus, $\Gamma = 100$. Suppose, for simplicity, that no task clustering is done. From the equations in Section 4.2, μ can be seen to be equal to 6. Therefore, $\phi = \lceil 100 \div (50 - 6) \rceil = 3$. Let the three instances of *T1* be labeled as $T1^1$, $T1^2$ and $T1^3$. The constituent tasks of *T1* are similarly labeled. The execution slots for the aperiodic task graph are allocated at {44, 50}, {88, 94} and {132 − 100, 138 − 100} = {32, 38}. Figure 3(b) shows a feasible architecture along with its task and edge allocation. Figure 3(c) shows the PE/link schedule for this architecture.

5 The CASPER Algorithm

In this section, we first provide an overview of CASPER and then follow up with details on each step. Figure 4 presents one possible co-synthesis process flow which we follow in our work. This flow is divided up into two parts: pre-processing and synthesis. During pre-processing, we process the task graph, system constraints and resource library, and create necessary data structures. In traditional real-time computing theory, if $period_i$ is the period of task graph i then [hyperperiod÷ $period_i$] copies are obtained for it [18]. However, this is impractical from both co-synthesis CPU time and memory requirements point of view, specially for multi-rate task graphs where this ratio may be very large. We tackle this problem by using the concept of *association array* [2]. The *clustering* step involves grouping of tasks to reduce the search space for the allocation step [2]. Tasks in a cluster get mapped to the same PE. This significantly reduces the overall complexity of the co-synthesis algorithm since allocation is part of its inner loop. At this point, an initial schedule length is derived for the aperiodic task graphs. Then clusters are ordered based on their importance/priority.

The synthesis step determines the allocation for both periodic and aperiodic task graphs. The synthesis part has two loops: 1) an *outer loop* for allocating each cluster, and 2) an *inner loop* for evaluating various allocations for each cluster. For each cluster, an *allocation array* consisting of the possible allocations at that step is created. While allocating a cluster to a hardware module such as an ASIC or FPGA, it is made sure that the module capacity related to pin count, gate count, *etc*., is not exceeded.

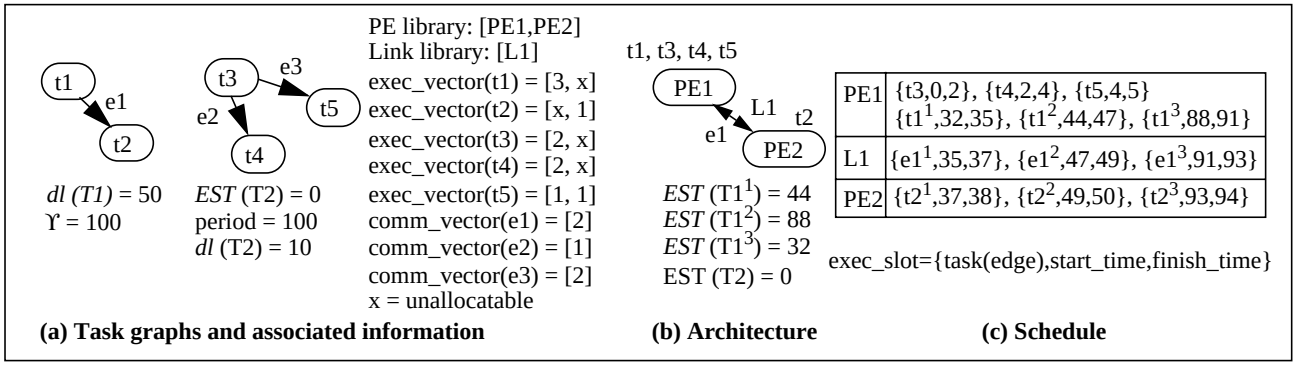

Figure 3. Scheduling of aperiodic and periodic task graphs with inter-task communication

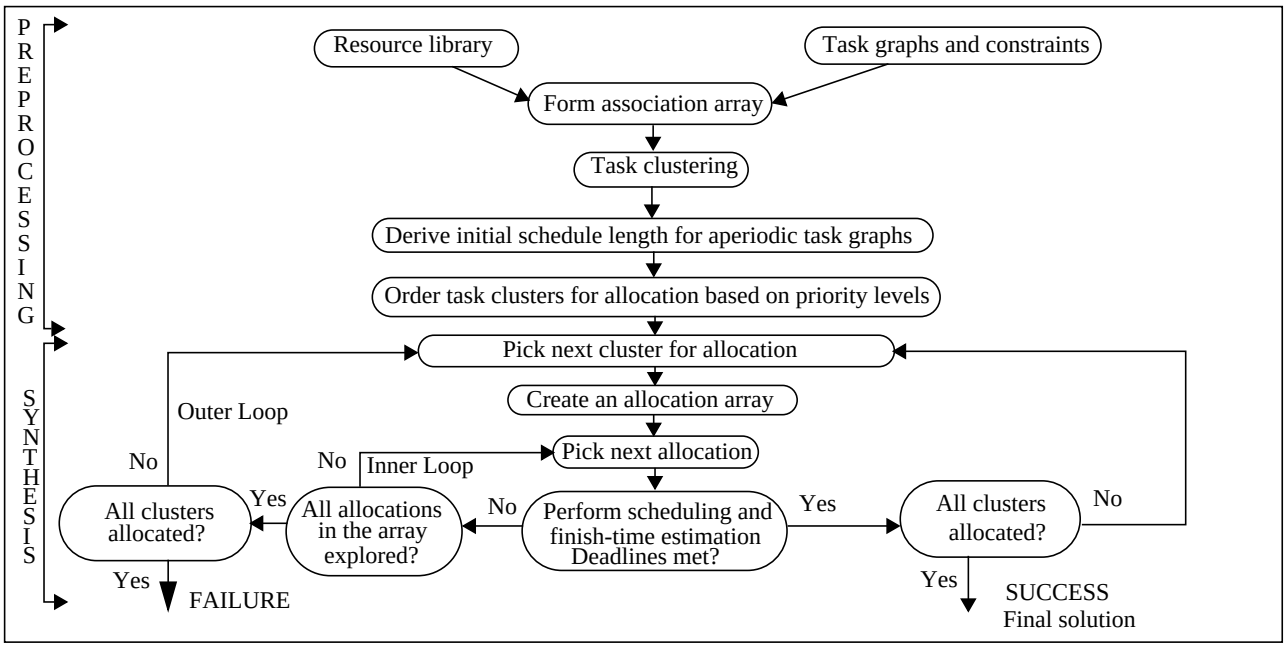

Figure 4. The co-synthesis process flow

Similarly, while allocating a cluster to a general-purpose processor, it is made sure that the memory capacity of the PE is not exceeded. Inter-cluster edges are allocated to resources from the link library.

The next step is *scheduling* which determines the relative ordering of tasks/edges for execution and the start and finish times for each task and edge. We employ a combination of both preemptive and non-preemptive static scheduling. Preemptive scheduling is used in restricted scenarios to minimize scheduling complexity (see Section 5.4). For task preemption, we take into consideration the operating system overheads such as interrupt overhead, context-switch, remote procedure call (RPC), *etc.*, through a parameter called preemption overhead (this information is experimentally determined and provided *a priori*). Incorporating scheduling into the inner loop facilitates accurate performance evaluation. *Performance evaluation* of an allocation is extremely important in picking the best allocation. An important step of performance evaluation is *finish-time estimation*. In this step, with the help of the scheduler, the finish times of each task and edge are estimated using the longest path algorithm [2]. After finish-time estimation, it is verified whether the given deadlines in the task graphs are met. The *allocation evaluation* step compares the current allocation against previous ones based on total dollar cost of the architecture.

5.1 The association array

Traditionally, as mentioned before, each task graph is replicated the requisite number of times in the hyperperiod. This is the approach used in the co-synthesis algorithms in [10]. We use the concept of association array [2] to eliminate the need for replication of task graphs in the hyperperiod. An association array contains limited information about each copy of the task graph. Our experience from COSYN [2] shows that up to 8-fold reduction in co-synthesis CPU time is possible for medium-sized task graphs (with tasks numbering in the hundreds) with less than 1% increase in system cost. It not only eliminates the need to replicate task graphs, but it also allows allocation of different task graph copies to different PEs, if desirable, to derive an efficient architecture. This array also supports pipelining of task graphs. This is explained next.

There are two types of periodic task graphs: 1) those with a deadline less than or equal to the period, and 2) those with a deadline greater than the period. In order to address this fact, the association array can have two dimensions. If a task graph has a deadline less than or equal to its period, it implies that there will be only one instance of the task graph in execution at any instant. Such a task graph needs only one dimension in the association array, called the horizontal dimension. If a task graph has a period less than its deadline, it implies that there can be more than one instance of this task graph in execution at some instant, *e.g.* MPEG frame processing. For such tasks, we create a two-dimensional association array, where the vertical dimension corresponds to concurrent execution of different instances of the task graph. For aperiodic task graphs, Υ is used akin to period for determining concurrent instances.

Concurrent instances of task graphs are allocated to the same set of PEs to achieve pipelining. For example, consider the aperiodic task graph, resource library and execution/ communication time vectors shown in Figure 5(a). Since its deadline is 90 and minimum inter-instance time interval is 30, three concurrent instances of the task graph may be running, as shown in Figure 5(b). These concurrent aperiodic task graphs could be allocated as shown in Figure 5(c) to achieve a pipelined architecture ($PE1^1$ and $PE1^2$ are two instances of the PE library element PE1).

Tasks, which do not start at $EST = 0$, may have the execution interval of their last copy exceed the hyperperiod. The portion of the execution interval, which exceeds the hyperperiod, is termed as *hyperperiod spill*. In order to ensure that the resulting schedule is feasible and resources are not overused, we must make space for the required hyperperiod spill at the beginning of the hyperperiod (since the schedule derived for a hyperperiod is repeated for successive hyperperiods). Hence, for such tasks we reassign their priority level by adding the hyperperiod to it (the concept of priority level is described in Section 5.2). Doing this gives such tasks much higher priority than other tasks in the system, enabling them to find a suitable slot at the beginning of the next hyperperiod. We use this reassigned priority level during scheduling. If the required spill is still not available after the priority level reassignment (this could be due to competing tasks which either required a spill or must start at the beginning of the hyperperiod), we upgrade the allocation.

5.2 Task clustering

Clustering involves grouping of tasks to reduce the complexity of allocation. Our clustering technique addresses the fact there may be multiple longest paths through the task graph

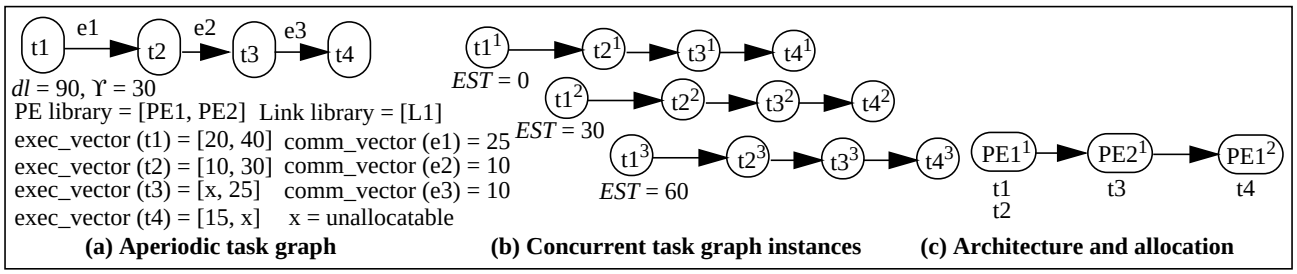

Figure 5. Task graph pipelining

and the length of the longest path changes after partial clustering. We use the critical path task clustering method given in [2]. In order to cluster tasks, we first assign deadline-based priority levels to tasks and edges using the procedure from [2]. The priority level of a task is an indication of the longest path from the task to a task with a specified deadline in terms of computation and communication costs as well as the deadline. In the beginning, when allocation is not defined, we sum up the maximum execution and communication times along the longest path and subtract the deadline from the sum to determine the priority levels. However, priority levels are recomputed after each allocation as well as task clustering steps. In order to reduce the schedule length, we need to decrease the length of the longest path. This is done by forming a cluster of tasks along the current longest path. This makes the communication costs along the path zero. Then the process can be repeated for the longest path formed by the yet unclustered tasks, and so on. Our experience from COSYN [2] shows that task clustering results in up to three-fold reduction in co-synthesis CPU time for medium-sized task graphs with less than 1% increase in system cost.

5.3 Cluster allocation

Once the clusters are formed, we need to allocate them to PEs. We define the priority level of a cluster as the maximum of the priority levels of the constituent tasks and incoming edges. Clusters are ordered based on decreasing priority levels. After the allocation of each cluster, we recalculate the priority level of each task and cluster. We pick the cluster with the highest priority level and create an allocation array. This is an array of the possible allocations for a given cluster at that point in co-synthesis. It is formed considering preference vectors, upgrade of PEs, upgrade of links, addition of PEs and links, *etc*. Limiting the number of PEs and links that can be added at any step helps keep the allocation array size at manageable levels. We order the allocations in the allocation array in the order of increasing value of dollar cost. Once the allocation array is formed, we use the inner loop of co-synthesis to evaluate the allocations from this array. During this loop, we pick the allocation with the least dollar cost and perform scheduling and allocation evaluation. If deadlines are met, we pick the next cluster, otherwise we repeat the process with another allocation from the allocation array.

5.4 Scheduling

To determine the order of scheduling, we prioritize tasks and edges based on the decreasing order of their priority levels. If two tasks (edges) have equal priority levels then we schedule the task (edge) with the shorter execution (communication) time first. While scheduling communication edges, the scheduler considers the mode of communication (sequential or concurrent) supported by the link and the processor. Though preemptive scheduling is sometimes not desirable due to the overhead associated with it, it may be necessary to obtain an efficient architecture. The preemption overhead, ξ, is determined experimentally considering the operating system overhead. It includes context switching and any other processor-specific overheads. To minimize scheduling complexity, preemption of a higher priority task by a lower priority task is allowed only in the case when the higher priority task is a sink task which will not miss its deadline, in order to minimize the scheduling complexity. For each aperiodic task, as explained before, we position the execution slots throughout the hyperperiod after scheduling the first execution slot. If the execution slot cannot be allocated at the required instant, we schedule it at the earliest possible time and reposition the remaining slots to ensure that the deadlines are always met.

5.5 Performance estimation

We use the finish-time estimation technique using a longest path algorithm from [2] to estimate the finish times of all tasks with specified deadlines and check whether their deadlines are met. The scheduler provides accurate information on the start and finish times of the allocated tasks and edges. This, in turn, makes our finish-time estimation method more accurate and minimizes false rejection of an allocation. We store the start as well as the finish times of each task and edge based on its best-possible as well as the worst-possible allocation. When a task or edge gets allocated, its start times converge to one number, so do its finish times.

5.6 Allocation evaluation

Each allocation is evaluated based on the total dollar cost which is the summation of the dollar cost of constituent PEs and links. We pick the allocation which at least meets the deadline in the best case. If no such allocation exists, we pick an allocation for which the summation of the best-allocation based finish times of all tasks with specified deadlines (recall that a task graph can have more than one task with a specified deadline) in all task graphs is maximum. This generally leads to the least-expensive architecture since a larger finish time usually corresponds to a less expensive architecture (note that we can always upgrade the architecture at a later step, if necessary, to meet real-time constraints). If there are more than one allocation which meet this criterion then to break the tie we choose the allocation for which the summation of the worst-allocation based finish times of all tasks with deadlines is maximum.

6 Experimental Results

CASPER is implemented in C++. It was run on various Bell Laboratories telecom transport system task graphs. These are large task graphs representing real-life field applications. The execution times for the tasks in these graphs were either experimentally measured or estimated based on existing designs. The general-purpose processors in the resource library had the real-time operating system, pSOS$^+$, running on them. The execution times included the operating system overhead. For results on these graphs, the PE library was assumed to contain Motorola microprocessors 68360, 68040, 68060 (each processor with and without a second-level cache), 11 ASICs, one XILINX 3195A FPGA, one ORCA 2T15 FPGA, and two optical transmitter and receiver modules. The link library was assumed to contain a 680X0 bus, a 1 Mb/s LAN, a 10 Mb/s LAN, a 6.176 Mb/s serial link supporting broadcast mode, and a 31 Mb/s serial link. Telecom embedded systems contain a mix of periodic and aperiodic task graphs. For the eight telecom examples considered next, on an average 30% of the tasks were aperiodic.

Table 1 shows the experimental results. The first major column in this table gives characteristics of the distributed architecture derived by CASPER employing the slack stealing [13] concept. In this case, hard aperiodic task graphs are allocated after the architecture for hard periodic task graphs is defined. Slacks from the schedules of the periodic task graphs

.

Table 1: Experimental results for telecom transport system examples

Example/ (No. of tasks)	CASPER with slack stealing				CASPER with RMS				CASPER					
	No. of PEs	No. of links	CPU time (sec.)	Cost ($)	No. of PEs	No. of links	CPU time (sec.)	Cost ($)	No. of PEs	No. of links	CPU time (sec.)	Cost ($)	Cost savings over slack stealing %	Cost savings over RMS %
CATS1/(112)	4	2	106.2	665	5	3	117.5	725	3	2	94.6	520	21.8	28.3
CATS2/(178)	7	4	187.1	1504	8	5	211.4	1635	5	3	136.2	1135	24.5	30.6
CATS3/(324)	10	4	1619.7	1965	12	5	1880.5	2355	8	4	1400.1	1580	19.6	32.9
CHAS1/(539)	15	11	1938.5	1945	14	11	2014.6	1830	12	10	1720.5	1460	24.9	20.2
CHAS2/(712)	22	9	4940.1	9080	24	10	5160.2	10865	18	7	4218.1	7128	21.5	34.4
CHAS3/(848)	19	9	7195.3	8070	22	11	7610.6	9150	15	6	5890.6	6560	18.7	28.3
CHAS4/(1018)	37	14	19956.3	13930	39	16	21060.1	14910	29	8	16928.3	9980	28.4	33.1
CHARS/(1241)	46	16	20390.2	16780	45	15	19830.1	17610	34	10	17140.2	12820	23.6	27.2

are stolen to service aperiodic task graphs, and the architecture is upgraded when necessary. The CPU times are on Sparcstation 20 with 256 MB of DRAM. The system cost is the summation of the costs of the constituent PEs and links. The second major column gives results for CASPER using RMS [17]. In this case, aperiodic and periodic task graphs are handled concurrently. For RMS, priority levels are assigned based on task graph periods, where task graphs with a shorter period receive higher priority. In case of an aperiodic task graph, its minimum inter-instance time interval is treated akin to the period for assigning the priority level. If two tasks (edges) have the same priority level, we schedule the task (edge) with the smaller execution (communication) time first. The third major column gives the results with CASPER employing the scheduler using deadline-based priority levels, and invoking concurrent co-synthesis of aperiodic and periodic task graphs.

CASPER realizes on an average (average of individual cost reductions) 22.9% architecture cost savings over the slack stealing algorithm and 29.4% over RMS.

When no aperiodic tasks are present, CASPER reduces to COSYN [2]. The efficacy of COSYN with respect to other co-synthesis systems which handle periodic task graphs has been established in [2].

7 Conclusions

We have presented an efficient co-synthesis algorithm for synthesizing distributed embedded system architectures for hard real-time aperiodic and periodic task graphs. Experimental results for various large real-life telecom system examples are very encouraging. We have also demonstrated the efficacy of using our scheduling technique in the co-synthesis algorithm as opposed to slack stealing or RMS. This is the first work to provide simultaneous support of periodic and aperiodic task graphs with hard deadlines during co-synthesis that provides a guarantee that the real-time constraints will always be met. Currently, CASPER is being applied to several next generation telecom transport system projects at Lucent Technologies.

References

1. M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Co., 1979.
2. B. P. Dave, G. Lakshminarayana, and N. K. Jha, "COSYN: Hardware-software co-synthesis of distributed embedded systems," in *Proc. Design Automation Conf.*, pp. 703-708, June 1997.
3. R. K. Gupta, *Co-Synthesis of Hardware and Software for Digital Embedded Systems*, Kluwer Academic Publishers, Norwell, Mass., 1995.
4. R. Ernst, J. Henkel and T. Benner, "Hardware-software co-synthesis for microcontrollers," *IEEE Design & Test of Computers*, vol. 10, no. 4, pp. 64-75, Dec. 1993.
5. K. Buchenrieder and C. Veith, "A prototyping environment for control-oriented HW/SW systems using state-charts, activity charts and FPGA's," in *Proc. European Design Automation Conf.*, pp. 60-65, Sept. 1994.
6. J. A. Rowson, "Hardware/software co-simulation," in *Proc. Design Automation Conf.*, pp. 439-440, June 1994.
7. S. Prakash and A. Parker, "SOS: Synthesis of application-specific heterogeneous multiprocessor systems," *J. Parallel & Distributed Comput.*, vol. 16, pp. 338-351, Dec. 1992.
8. J. G. D'Ambrosio and X. Hu, "Configuration-level hardware/software partitioning for real-time systems," in *Proc. Int. Wkshp. Hardware-Software Co-Design*, pp. 34-41, Sept. 1994.
9. T.-Y. Yen and W. Wolf, "Communication synthesis for distributed embedded systems," in *Proc. Int. Conf. Computer-Aided Design*, pp. 288-294, Nov. 1995.
10. D. Kirovski and M. Potkonjak, "System-level synthesis of low-power hard real-time systems," in *Proc. Design Automation Conf.*, pp. 697-702, June 1997.
11. K. Ramamritham and J. A. Stankovic, "Scheduling algorithms and operating systems support for real-time systems," *Proc. IEEE*, Jan. 1994.
12. J. Lehoczky and S. Ramos-Thuel, "An optimal algorithm for scheduling soft aperiodic tasks in fixed-priority preemptive systems," in *Proc. Real-Time Systems Symp.*, pp. 110-123, Dec. 1992.
13. R. I. Davis, K. W. Tindell, and A. Burns, "Scheduling slack time in fixed priority pre-emptive systems," in *Proc. Real-Time Systems Symp.*, pp. 222-231, Dec. 1993.
14. K. Ramamritham, J. A. Stankovic, and P. Shiah, "Efficient scheduling algorithms for real-time multiprocessor systems," *IEEE Trans. Parallel & Distributed Systems*, pp. 184-194, Apr. 1990.
15. K. S. Hong and J. Y.-T. Leung, "On-line scheduling of real-time tasks," in *Proc. Real-Time Systems Symp.*, pp. 244-250, Dec. 1988.
16. B. Kao and H. Garcia-Molina, "Deadline assignment in a distributed soft real-time system," *Tech. Rep.*, Stanford University, STAN-CS-92-1452, Oct. 1992.
17. L. Sha *et al.*, "Generalized rate-monotonic scheduling theory: A framework for developing real-time systems," *Proc. IEEE*, Jan. 1994.
18. E. Lawler and C. Martel, "Scheduling periodically occurring tasks on multiple processors," *Information Processing Letters*, vol. 12, pp. 9-12, Feb. 1981.