COSYN: Hardware–Software Co-Synthesis of Heterogeneous Distributed Embedded Systems

Bharat P. Dave, *Member, IEEE,* Ganesh Lakshminarayana, and Niraj K. Jha, *Fellow, IEEE*

***Abstract*— Hardware–software co-synthesis starts with an embedded-system specification and results in an architecture consisting of hardware and software modules to meet performance, power, and cost goals. Embedded systems are generally specified in terms of a set of acyclic task graphs. In this paper, we present a co-synthesis algorithm COSYN, which starts with periodic task graphs with real-time constraints and produces a low-cost heterogeneous distributed embedded-system architecture meeting these constraints. It supports both concurrent and sequential modes of communication and computation. It employs a combination of preemptive and nonpreemptive static scheduling. It allows task graphs in which different tasks have different deadlines. It introduces the concept of an association array to tackle the problem of multirate systems. It uses a new task-clustering technique, which takes the changing nature of the critical path in the task graph into account. It supports pipelining of task graphs and a mix of various technologies to meet embedded-system constraints and minimize power dissipation. In general, embedded-system tasks are reused across multiple functions. COSYN uses the concept of architectural hints and reuse to exploit this fact. Finally, if desired, it also optimizes the architecture for power consumption. COSYN produces optimal results for the examples from the literature while providing several orders of magnitude advantage in central processing unit time over an existing optimal algorithm. The efficacy of COSYN and its low-power extension COSYN-LP is also established through their application to very large task graphs (with over 1000 tasks).**

***Index Terms*— Allocation, embedded systems, hardware-software co-synthesis, low power, scheduling, system synthesis.**

I. Introduction

ADVANCEMENTS in very large scale integration (VLSI), computer-aided design, and packaging areas have resulted in an explosive growth in embedded systems. Heterogeneous distributed architectures are common for such systems, where several processors, application-specific integrated circuits (ASIC's), and field-programmable gate arrays (FPGA's) are interconnected by various types of communication links, and multiple tasks are concurrently run on the system. Each task can be executed on a variety of software and hardware platforms with different dollar costs. Finding an optimal hardware–software architecture entails selection of processors, ASIC's, FPGA's, and communication links such that the architecture cost is minimum and all real-time constraints are met. For low-power embedded systems, the aim is to obtain an architecture with minimum average power dissipation while meeting all real-time and peak power constraints. Hardware–software co-synthesis involves allocation, scheduling, and performance estimation. Allocation determines the mapping of tasks to processing elements (PE's) and inter-task communications to communication links. Scheduling determines the sequencing of tasks mapped to a PE and communications on a link. Performance estimation determines the finish time of each task in the embedded-system specification and the quality of the system in terms of dollar cost, power consumption, fault tolerance, etc. Both allocation and scheduling are known to be NP complete [1]. Therefore, optimal co-synthesis is computationally hard.

Emphasis on distributed embedded-system architecture co-synthesis and partitioning is fairly recent [2]–[12]. The optimal co-synthesis approaches are mixed-integer linear programming (MILP) [7] and exhaustive [8]. These are, however, applicable to very small task graphs (consisting of ten or so tasks). The heuristic approaches are iterative and constructive. In the former, an initial solution is iteratively improved through various moves. In the latter, the solution is built step-by-step. The iterative procedure in [9] and [10] considers only one type of communication link and does not allow mapping of each successive copy of a periodic task to a different PE. The iterative synthesis technique for low-power systems in [11] ignores inter-task communications, and is restricted to periodic task graphs for which the deadline is equal to the period. The constructive fault tolerance procedure in [12] does not support communication topologies such as bus, local area network (LAN), etc., and is not suitable for multirate embedded systems, e.g., multimedia systems.

We have developed a heuristic-based constructive co-synthesis technique called COSYN, which includes allocation, scheduling, and performance estimation steps as well as power optimization features. COSYN takes as an input periodic acyclic task graphs and generates a low-cost heterogeneous distributed embedded-system architecture meeting real-time constraints. It is suited to both small- and large scale real-time embedded systems. Application of this technique to several examples from the literature and real-life telecom transport systems shows that it compares very favorably with known co-synthesis algorithms in terms of central processing unit (CPU) time, quality of solution, and number of features.

Manuscript received September 24, 1997; revised February 20, 1998 and June 10, 1998.This work was supported in part by Bell Laboratories, Lucent Technologies and in part by the National Science Foundation under Grant MIP-9423574.

B. P. Dave is with Bell Laboratories, Lucent Technologies, Holmdel, NJ 07733 USA.

G. Lakshminarayana is with Computer and Communications Research Laboratories (CCRL), NEC, Princeton, NJ 08540 USA.

N. K. Jha is with the Department of Electrical Engineering, Princeton University, Princeton, NJ 08544 USA.

Publisher Item Identifier S 1063-8210(99)01550-4.

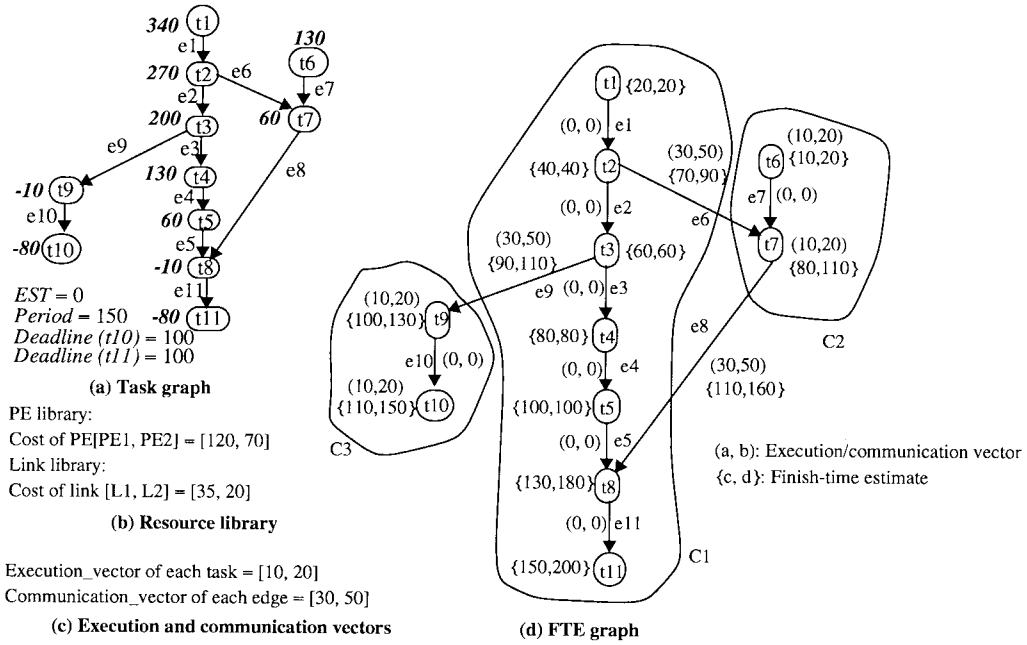


Fig. 1. Task graph, resource library, execution/communication vectors, and the finish-time estimation (FTE) graph.

The paper is organized as follows. Section II provides preliminaries. Section III describes the steps of our co-synthesis algorithm. Section IV describes the low-power extension. Section V gives experimental results. Section VI gives the conclusions.

II. Preliminaries

In this section, we give the basic definitions and concepts which form the basis for the co-synthesis framework.

A. Task Graphs

Each application-specific function is made up of several sequential and/or concurrent *jobs*. Each job is made up of several tasks. A task contains both data- and control-flow information. The embedded system 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 between them. In this paper, we focus on periodic task graphs with real-time constraints. Each periodic task graph has an earliest start time (*EST*), period, and deadlines, as shown, for an example, in Fig. 1(a). Each task of a periodic task graph inherits the graph's period and can have a different deadline.

B. Definitions and Basic Concepts

The PE (link) library is a collection of all available PE's (communication links). The PE and link libraries together form the *resource library*. The resource library and its costs for two general-purpose processors, *PE1* and *PE2*, and two links, *L1* and *L2*, are shown in Fig. 1(b).

We define execution_vector$(t_i) = \{\alpha_{i1}, \alpha_{i2}, \cdots, \alpha_{\text{in}}\}$ where α_{ij} indicates the execution time of task t_i on PE j from the PE library. $\alpha^{\min}(t_i)$ $(\alpha^{\max}(t_i))$ denote the minimum (maximum) entries in this vector. In Fig. 1(c), for simplicity, all tasks are assumed to have the same execution vector. Execution vectors are derived from laboratory measurements or simulation or through worst-case delay estimation tools [13]. We define preference_vector$(t_i) = \{\gamma_{i1}, \gamma_{i2}, \cdots, \gamma_{\text{in}}\}$ where γ_{ij} indicates preferential mapping for task t_i. If γ_{ij} is 0, t_i cannot be executed on PE j, and 1 if there are no constraints. Similarly, we define exclusion_vector$(t_i) = \{\delta_{i1}, \delta_{i2}, \cdots, \delta_{iq}\}$ where $\delta_{ij} = 1$ indicates that tasks t_i and t_j have to be allocated to different processors, and $\delta_{ij} = 0$ indicates otherwise.

A cluster of tasks is a group of tasks all of which are allocated to the same PE. For example, Fig. 1 (d) shows three clusters: $C1$, $C2$, and $C3$. We define preference_vector(C_i) of cluster C_i to be the bit-wise logical AND of the preference vectors of all the tasks in the cluster. This vector indicates which PE's the cluster cannot be allocated to. Similarly, we define exclusion_vector(C_i) of cluster C_i to be the bit-wise logical OR of the exclusion vectors of all the tasks in the cluster. Task t_j is said to be preference-compatible with cluster C_i if the bit-wise logical AND of the preference vector of cluster C_i and task t_j does not result in a vector with all elements zero. A zero-vector makes the cluster unallocatable to any PE. Task t_j is said to be exclusion-compatible with cluster C_i if the jth entry of the exclusion vector of C_i is zero. This indicates that tasks in cluster C_i can be co-allocated with task t_j. Task t_j and cluster C_i are simply called *compatible*, if t_j is both preference- and exclusion-compatible with cluster C_i.

We define communication_vector$(e_k) = (\beta_{k1}, \beta_{k2}, \cdots, \beta_{km})$ where β_{kl} indicates the time it takes to communicate the data on edge e_k on communication link l from the link library. The communication vector for each edge in the task graph of Fig. 1(a) is given in Fig. 1(c). $\beta^{\min}(e_k)$ $(\beta^{\max}(e_k))$ denote the minimum (maximum) entries in this vector. This vector is computed *a priori* for various types of links as follows. Let ρ_k be the number of bytes that need to be communicated on edge e_k, and λ_l be the number of bytes per packet that link l can support, excluding the packet overhead.

We define access_time_vector$(l) = [\Omega_{l1}, \Omega_{l2}, \cdots, \Omega_{lm}]$ where Ω_{lr} represents the access time per packet with r number of communication ports on link l. Suppose the link under consideration l has s ports. Let τ_l be the communication time of a packet on link l. Then

$$\beta_{kl} = \{\lceil(\rho_k) \div (\lambda_l)\rceil \bullet (\tau_l + \Omega_{ls})\}.$$

This equation is applicable to a link which requires an access time overhead for each packet. However, some links support a burst mode of communication, where multiple packets can be transmitted with only a one-time link access overhead. Thus, this overhead can be reduced in such cases. Initially, since the actual number of communication ports on the links is not known, we use an average number of communication ports (specified *a priori*) to determine the communication vector. This vector is recomputed after each allocation, considering the actual number of ports on the link.

We define average_power_vector$(t_i) = \{\xi_{i1}, \xi_{i2}, \cdots, \xi_{\text{in}}\}$ where ξ_{ij} indicates the average power consumption of task t_i on PE j. Similarly, we define peak_power_vector$(t_i) = \{\kappa_{i1}, \kappa_{i2}, \cdots, \kappa_{\text{in}}\}$. Preference, exclusion, average and peak power vectors can be similarly defined for communication edges and links. We also take into account the quiescent power of a PE, link, ASIC and FPGA, which indicates its power consumption at times when no task (or communication) is being executed on it.

The storage requirements are of different types: program storage, data storage, and stack storage. For each task mapped to software, memory needs are specified by a *memory vector*. The memory vector of task t_i is defined as: memory_vector(t_i) = [program_storage(t_i), data_storage(t_i), stack_storage(t_i)].

For each available processor, its cost, supply voltage, average quiescent power consumption, peak power constraint, and attributes such as memory architecture, number of communication ports, processor-link communication, and cache characteristics are assumed to be specified. Also, the preemption overhead for each processor is specified *a priori* along with its execution time and average and peak power consumption. For each ASIC, its cost, supply voltage, available pins, available gates, and average and peak power dissipation per gate are assumed to be specified. For each FPGA, its cost, supply voltage, average quiescent power, available pins, and the maximum number of flip-flops or combinational logic blocks (CLB's) or programmable functional units (PFU's) are assumed to be specified. The boot memory also needs to be allocated for the FPGA. Generally, all flip-flops/CLB's/PFU's are not usable due to routing restrictions. Based on our experience, we assume only 70% (this percentage is user-specifiable) are actually usable. The user can also specify the percentage of package pins that can be used for allocation (default is 80% to allow for pins for power, ground, and due to routing restrictions).

Generally, several tasks are reused across multiple functions. To exploit this fact, architectural hints are derived during task graph generation based on prior experience, nature of embedded-system task graphs, and type of resource library. If the hint marks the task or sub-task-graph for reuse, the co-synthesis algorithm is run for each such task/sub-task-graph and the solution is stored as an architectural template. During allocation, if such a task/sub-task-graph is being considered, then, if necessary (the template may already be in the partial architecture), we add the architectural template to the partial architecture and proceed further.

```
COSYN(task graphs, resource library, constraints){
  assign deadline-based priority levels(task graph);
  FORM_ASSOCIATION_ARRAY(task_graphs);
  cluster_list ← FORM_CLUSTERS(task graphs);
  sorted_cluster_list ←sort clusters in the order
  of decreasing priority levels;
  for each cluster in sorted_cluster_list{
    cluster_tag=UNALLOCATED;}
  partial_architecture = NULL;
  ALLOCATE_CLUSTERS(sorted_cluster_list,
  partial_architecture){
  for each unallocated cluster C_i from sorted_cluster_list {
    allocation array ← FORM_ALLOC_ARRAY(C_i,
    partial_architecture);
    for each allocation in allocation array {
      schedule allocated clusters;
      performance evaluation;// performs finish time,
      //energy, and power estimation
      if (deadline met in the best case) {
        partial_architecture = current_allocation;
        break;}
      else {
        partial_architecture = best allocation;}}
    cluster_tag(C_i) = ALLOCATED;}}
  return final_solution = partial_architecture;}
```

Fig. 2. The COSYN procedure.

III. The COSYN Algorithm

We next provide an overview of COSYN, followed by details. Fig. 2 presents the pseudo-code for COSYN. First, task graphs, system/task constraints, and resource library are parsed and appropriate data structures created. The hyperperiod of the system is computed as the least common multiple (LCM) of the period of various task graphs. Traditionally, if $period_i$ is the period of task graph i then [hyperperiod $\div$ $period_i$] copies are obtained for it [14]. However, this is impractical from both co-synthesis CPU time and memory requirements point-of-view, specially for multirate task graphs or task graphs with co-prime periods where this ratio may be very large. We tackle this problem using the concept of an *association array*. *Task clustering* involves grouping of tasks to reduce the search space for allocation [12]. Tasks in a cluster get mapped to the same PE. Clusters are ordered based on their priority. The *outer loop* of a procedure selects a cluster, and the *inner loop* evaluates various allocations for each selected cluster. For each cluster, an *allocation array* consisting of the possible allocations is created. Inter-cluster edges are allocated to resources from the link library.

We employ a combination of preemptive and nonpreemptive static scheduling. We take into account the operating system overheads, e.g., interrupt overhead, context-switch, remote procedure call (RPC), etc., through a parameter called preemption overhead. Incorporating scheduling into the inner loop facilitates accurate performance evaluation. As part of

performance evaluation, FTE uses the longest path algorithm to check whether specified deadlines of tasks are met. The *allocation evaluation* step compares the current allocation against previous ones based on total dollar cost. If there is more than one allocation with equal dollar cost, we pick the allocation with the lowest average power consumption. Memory requirements and peak power dissipation can also be used to further evaluate the allocations. Next, we describe each step of COSYN in detail.

A. The Association Array

We use two approaches to tackle the problem of large number of task copies.

In the first approach, we shorten some of the periods by a small user-adjustable amount (up to 3% used as a default) to reduce the hyperperiod [14]. This is frequently useful even if the periods are not co-prime, but the hyperperiod is large. Doing this usually does not affect the feasibility of co-synthesis or the architecture cost. We first identify the task graphs which require a large number of copies. We rank such task graphs in the order of decreasing number of copies. We pick the highest ranked task graph and adjust its period up to the user-defined threshold, and check its impact on the number of copies for the other task graphs. We proceed down the rank and stop when we bring the number of required copies below a specified threshold.

In the second approach, we use the concept of association array to avoid the actual replication of task graphs. This array has an entry for each task of each copy of the task graph such as the PE to which it is allocated, its priority level, deadline, best-case projected finish time (PFT), and worst-case PFT. The deadline of the nth instance of a task is offset by $(n-1)$ multiplied by its period from the deadline in the original task. This concept allows allocation of different task graph copies to different PE's, if desirable, to derive an efficient architecture. It also supports pipelining of task graphs, as explained later.

If a task graph has a deadline less than or equal to its period, there will be only one instance of the task graph in execution at any instant. Such a task graph needs only the horizontal dimension in the association array. If a task graph has a deadline greater than the period, there can be more than one instance of this task graph in execution at some instant. For such tasks, the vertical dimension corresponds to concurrent execution of different instances of the task graph.

In preprocessing, for each task graph with a deadline greater than its period, we find the association array depth and the modified period. The depth is given by $\lceil$(deadline of task graph $\div$ period)$\rceil$ and the modified period by the depth of the array multiplied by the original period. A task graph with a deadline less than or equal to its period does not require any modification of the period. The hyperperiod is computed based on the modified periods. The rows of the association array represent the concurrent existence of different instances of the task graph. Each row inherits the modified period of the task graph. The columns of the association array represent various copies of the task graph in the hyperperiod. Fig. 3 gives the pseudo-code for the association array formation procedure.

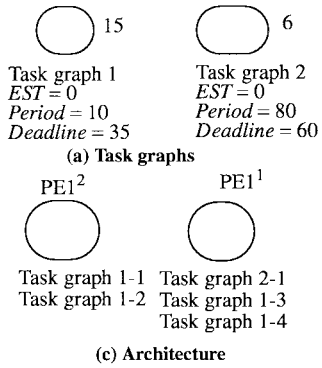
```
FORM_ASSOCIATION_ARRAY (task graphs){
  for each task graph T_i {
    //Each task graph has a deadline equal to the
    //maximum of all specified deadlines in it
    if (period < deadline){
      T_i _depth=⌈(T_i _deadline ÷ T_i _period)⌉
      T_i _modified_period = T_i _depth • T_i _period;}
    else{
      T_i _depth=1;
      T_i _modified_period = T_i _period;}}
  hyperperiod ← LCM of modified_periods of all
  task graphs
  for each task graph T_i {
    T_i _length=hyperperiod ÷ modified_period;}
  for each task t_j{
    create an association array of dimensions
    (t_j_task_graph_length, t_j_task_graph_depth);
    INITIALIZE_ASSOCIATION_ARRAY(task t_j);}}
INITIALIZE_ASSOCIATION_ARRAY(task t_j){
  T_k =t_j _task graph;
  for i =1 to T_k _depth{
    for j =1 to T_k _length{
      EST(i, j) = T_k _EST + ((i −1) • T_k _period) +
      ((j −1) • T_k _modified_period);
      deadline(i, j) = EST(i, j) + T_k _deadline;
      //start and finish times are updated by the
      //scheduling & finish-time estimation procedures.
      best_case_start(i,j)= worst_case_start(i,j)= − ∞;
      best_case_finish(i,j)= − ∞;
      worst_case_finish(i,j)= − ∞;}}}
```

Fig. 3. The association array formation procedure.

Copy 1 of the task in the first row inherits the *EST* and deadline from its task graph. If there are multiple tasks with different deadlines in the original task graph, then each task in the association array inherits the corresponding deadline from the task graph. The best- and worst-case start and finish times of the first copy are determined through scheduling and FTE. For the remaining copies, all parameters are set based on those of the first copy, e.g., the *EST* of the nth copy $=$ [the *EST* of copy $1 + (n-1)$ • *modified period*].

As an example, consider task graphs 1 and 2, shown in Fig. 4(a). For simplicity, assume that there is only one task in each task graph and only one type of PE is available. The execution times of the tasks on this PE are shown next to the corresponding nodes. There could be up to four instances of task graph 1 executing at any instant. The modified period of task graph 1 is 40 and the hyperperiod of task graphs 1 and 2 is 80. The association array for task graphs 1 and 2 is shown in Fig. 4(b). The start time of concurrent instances of task graph 1 is staggered by its period 10. The start time of the second copy of each instance is offset by the modified period. The deadline of the second copy of task graph 1-1 is set equal to the sum of its start time and corresponding deadline, i.e., $40 + 35 = 75$. The priority level of each task is calculated by subtracting its deadline from its execution time, as explained later. Hence, the priority level of copy 2 of task graph 1-1 is equal to $15 - 75 = -60$. Fig. 4(c) illustrates the associated architecture.

Another limitation of Lawler and Martel's approach [14] is that the execution of all copies of all tasks must complete by the hyperperiod. However, this puts significant restrictions on the scheduler. Tasks, which do not start at *EST* $= 0$,



Task	[PE instance, EST, deadline, priority level, best-case PFT, worst-case PFT]	
	Copy 1	Copy 2
Task graph 1-1	[PE1[1], 0, 35, -20, 15, 15]	[PE1[1], 40, 75, -60, 55, 55]
Task graph 1-2	[PE1[1], 10, 45, -30, 30, 30]	[PE1[1], 50, 85, -70, 70, 70]
Task graph 1-3	[PE1[2], 20, 55, -40, 35, 35]	[PE1[2], 60, 95, -80, 75, 75]
Task graph 1-4	[PE1[2], 30, 65, -50, 50, 50]	[PE1[2], 70, 105, -90, 90, 90]
Task graph 2-1	[PE1[2], 0, 60, -54, 16, 16]	Not Applicable

(b) Association array

Fig. 4. A two-dimensional association array.

may have the execution interval of their last copy exceed the hyperperiod. To address this problem, the deadline of the last copy of the task graph can be set equal to the hyperperiod. However, this approach generally results in an increase in system cost. To address this concern, we use the concept of hyperperiod spill, which is the portion of the execution interval which exceeds the hyperperiod. 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, we enhance the priority level of such tasks by adding the hyperperiod to it. 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. If the required spill is still not available after the priority-level enhancement (this could be due to competing tasks which either require a spill or must start at the beginning of the hyperperiod), we upgrade the allocation. This approach is used for scheduling the second copy of task graph 1-4 in Fig. 4(b). Copy 2 of task graph 1-4 requires a hyperperiod spill of ten time units. The priority level of this copy is -90 and that of task graph 2-1 copy 1 is -54. The priority level of the former is enhanced by adding the hyperperiod to its priority level, i.e., $-90+80=-10$. Thus, this copy now has a higher priority level than that of task graph 2-1 copy 1. Therefore, the required hyperperiod spill is allocated at the beginning of the hyperperiod. Hence, task graph 2-1 starts execution at time unit 10 instead of time unit 0 and completes by time unit 16.

When possible, concurrent instances of task graphs are allocated to the same set of PE's and links to achieve pipelining. For example, consider the periodic task graph, resource library, and execution/communication vectors, shown in Fig. 5(a). Since its deadline is 60 and period is 15, four concurrent instances of the task graph may be running, as shown in Fig. 5(b). These concurrent periodic task graphs could be allocated, as shown in Fig. 5(c), to achieve a pipelined architecture. $L1^1$, $L1^2$ and $L1^3$ are different instances of link $L1$. Pipelining is done for the task graph which requires concurrent execution. This is either determined by architectural hints or based on its period and deadline. The pipeline stage size is controlled by a user-specified parameter called pipeline_threshold (default is equal to the period of the task graph). It is used

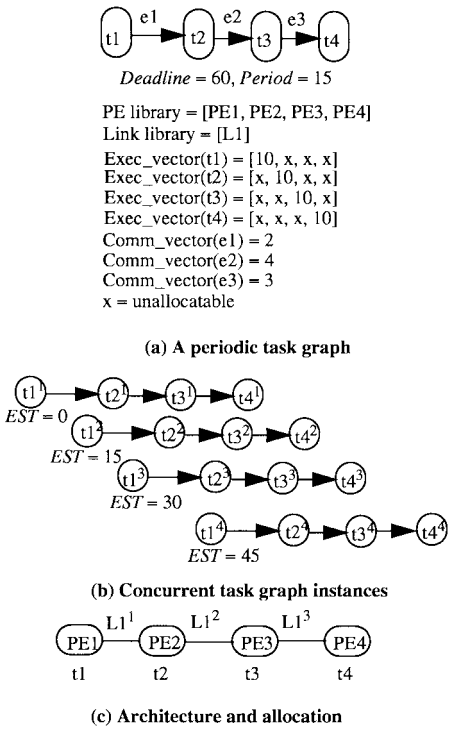


Fig. 5. Task-graph pipelining.

in a manner similar to the way the cluster size threshold is used during task clustering, as explained in the next section. The allocation of various pipeline stages is done during the allocation step by creating an allocation array. The same stages of different concurrent instances of task graphs are allocated to the same PE.

B. Task Clustering

Our clustering technique addresses the fact that different paths may become the longest path through the task graph at different points in the clustering process since the length of the longest path changes after partial clustering. We extend the method given in [12] for this purpose. Our procedure also supports task graphs in which different tasks have different deadlines. We first assign deadline-based priority levels to tasks and edges using the following procedure. A sink task always has a specified deadline, whereas a nonsink task may

```
FORM_CLUSTERS(task graphs){
  sorted_list ← sort tasks by decreasing priority levels;
  threshold = hyperperiod;
  for each task t_i {
    t_i_tag = NOT_CLUSTERED;}
  cluster_list ←NULL;
  for each unclustered task t_i from sorted_list {
    fan-in-set (t_i) = NULL;
    for each fan-in task t_j {
      if (t_j is not already clustered with another
      fanout task and its cluster is compatible with t_i){
        if (size (cluster(t_j) U t_i) ≤ hyperperiod){
          fan-in-set (t_i ) = fan-in-set(t_i) + t_j;}}}
    if (fan-in-set(t_i) = NULL){
      allocate a new cluster C_j;
      cluster_list ←cluster_list U (C_j);
      CLUSTER_GROWTH(C_j, t_i);}
    else{
      t_k← a task from fan-in-set(t_i) with the
      highest priority level along task t_i;
      C_j = cluster(t_k);
      β(t_k, t_i) = 0;
      CLUSTER_GROWTH(C_j, t_i);}
    sorted_list ← sort tasks by decreasing priority levels;}
  return cluster_list;}
```

Fig. 6. The critical path-based clustering procedure.

```
CLUSTER_GROWTH(Cluster C_k, task t_i){
  t_k = last task of cluster C_k;
  C_k = C_k U (t_i);// add task t_i to cluster C_k
  //update preference and exclusion vectors of cluster C_k
  preference_vector(C_k) = preference_vector(C_k) AND
  preference_vector (t_i);
  exclusion_vector(C_k) = exclusion_vector(C_k) OR
  exclusion_vector(t_i);
  t_i_tag = CLUSTERED;
  if (C_k has more than one task) {
    assign priority levels (t_i_task_graph);}
  if (size of cluster C_k is < hyperperiod){
    find the fan-out-set of t_i among its unclustered fanout
    tasks which are compatible with cluster C_k;
    if (fan-out-set is not empty) {
      eligible_fan_out_task t_m ← a task from the
      fan-out-set along the edge from t_i to which the
      priority level of t_i is the highest and the size of
      (C_k U t_m) is < hyperperiod;}
    if (t_m exists){
      β(t_i, t_m) = 0;
      C_k = C_k U {t_m};// add task t_m to cluster C_k
      //update preference and exclusion vectors
      //of cluster C_k
      preference_vector(C_k) = preference_vector(C_k)
      AND preference_vector (t_m);
      exclusion_vector(C_k) = exclusion_vector(C_k)
      OR exclusion_vector(t_m);
      t_m_tag = CLUSTERED;
      assign priority levels (t_i_task graph);}}
  return cluster_list;}
```

Fig. 7. The cluster growth procedure.

or may not. We define $\omega(t_j)$ to be equal to the deadline of task t_j if the deadline is specified, and ∞ otherwise.

1) Priority level of sink task $t_i = \alpha^{\max}(t_i) - \text{deadline } (t_i)$.
2) Priority level of edge $e_k =$ priority level of destination node $(e_k) + \beta^{\max}(e_k)$.
3) Priority level of nonsink task $t_j = \max$ (priority level of its fan-out edge e_f, $-\omega(t_j)) + \alpha^{\max}(t_j)$.

As an example, the numbers adjacent to the nodes in Fig. 1(a) indicate their associated priority levels. 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 and the deadline. To reduce the schedule length, we need to decrease the length of the longest path by clustering of tasks along it to make the communication costs along the path zero (this is based on the traditional assumption made in distributed computing that intra-PE communication takes zero time). Then the process can be repeated for the longest path formed by the yet unclustered tasks, and so on.

To ensure load balancing among various PE's, the cluster size is limited by a parameter called cluster-size threshold C_{th}. C_{th} is set equal to the hyperperiod Γ. Let there be k PE's in the PE library to which cluster C_k is allocatable. Thus, preference_vector(C_k) will have 1's corresponding to these k PE's. For any cluster C_k containing m tasks $\{t_1, t_2, \cdots, t_m\}$, its size, denoted as θ_k, is estimated as follows. Let p denote the period of the tasks in cluster C_k and let Γ be the hyperperiod. Then

$$\theta_k = \max_j \sum_{i=1}^{m} \alpha_{ij} \cdot (\Gamma \div p).$$

To take into consideration the worst-case allocation, we obtain θ_k as the maximum over all PE's of the summation of the execution times of all copies of all tasks in cluster C_k. Fig. 6 gives the critical path-based clustering procedure. Initially, we sort all tasks in the order of decreasing priority levels. We pick the unclustered task t_i with the highest priority level and mark it clustered. Then we find the fan-in set of t_i, which is a set of fan-in tasks that meet the following constraints:

1) the fan-in task is not clustered with another fan-out task;
2) the fan-in task's cluster C_k is compatible with t_i;
3) the size of cluster C_k does not exceed C_{th}.

If the fan-in set of t_i is not empty, we identify an eligible cluster which is grown (i.e., expanded) using the cluster growth procedure given in Fig. 7. If the fan-in set of t_i is empty, we allocate a new cluster C_j, and use the cluster growth procedure to expand it.

The cluster growth procedure adds task t_i to the feasible cluster identified from the fan-in set or to a new cluster, and grows the cluster further, if possible, by adding one of the compatible fan-out tasks of t_i along which the priority level of t_i is the highest. It recomputes the priority levels of the tasks in the task graph of t_i after clustering t_i either with any existing cluster or after clustering it with one of its fan-out tasks. This lets us identify and compress the critical path as it changes.

Consider the task graph in Fig. 8. For simplicity, assume that the resource library contains only one PE and one link. The execution and communication times are given in nonbold numbers next to nodes and edges, respectively. The deadline of this task graph is 34. The bold numbers indicate the initial priority levels of tasks and edges. Application of the clustering procedure from [12] results in two clusters: $(t1, t3, t4)$ and $(t2)$. The resulting architecture consists of two identical PE's connected with a link. The PFT with this architecture is

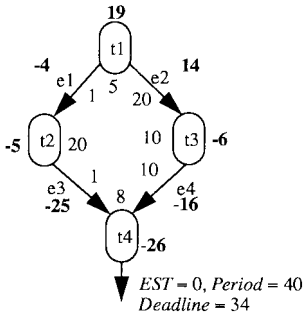


Fig. 8. A task graph to illustrate clustering.

35, which exceeds the deadline. However, our clustering procedure starts with task t_1, which has the highest priority level, and groups it with task t_3 since t_1 has a higher priority level along the edge to task t_3. At this point, the communication time of edge $e2$ is made zero and the priority levels recomputed. Since task t_2 now has a higher priority level than task t_3, clustering starts afresh from t_2. The resultant clusters are $(t1, t3)$ and $(t2, t4)$. The PFT is 34 with the two PE/one-link architecture, which meets the deadline. A task around which either a new cluster is formed or expanded is termed as the *seed* of the cluster. In [12], the seed is always the task that has just been clustered. In our method, we look for the best seed at each clustering step, giving it an advantage. Clustering of tasks can change the priority levels of the remaining tasks in the task graph. Therefore, it may impact the critical path. Although not illustrated by the example in Fig. 8, recomputing priority levels gives our method an additional advantage in accurately identifying a critical path, while taking into account the impact of clustering on priority levels of unclustered tasks.

The application of the clustering procedure to the task graph of Fig. 1(a) results in three clusters, *C1*, *C2*, and *C3*, as shown in Fig. 1(d). Once the clusters are formed, some tasks are replicated in two or more clusters to address inter-cluster communication bottlenecks [15]. This is useful when the increase in computation time is less than the decrease in the communication time. We replicate only those tasks which are compatible with the cluster. During the allocation step, if the two clusters are allocated to the same PE, then the replicated tasks are no longer needed to address the communication bottleneck(s). In that case, they are removed from the clusters.

C. Cluster Allocation

Once the clusters are formed, we need to allocate them. We define the priority level of a cluster to be 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. We order the allocations in this array in increasing cost order. We then use the inner loop of co-synthesis to evaluate the allocations.

An allocation array is created using the procedure given in Fig. 9. Architectural hints are used to pre-store allocation templates. We allow the addition of only two new PE's and

```
FORM_ALLOC_ARRAY(cluster C_i, partial architecture){
  form allocation array considering the following;
    1) addition of architectural templates based
       on architectural hints
    2) preference vector
    3) existing partial architecture
    4) upgrade of existing links
    5) upgrade of existing PEs
    6) addition of new PEs of each type
    7) addition of new links of each type to each PE
  for each allocation {
    for each PE and link{
      if (gate count, power, pin count, memory limits, or
      communication port count etc. are exceeded){
        remove allocation from the allocation array;
        break;}}}
  Order allocations in the order of increasing dollar cost;
  return allocation array; }
```

Fig. 9. The allocation array formation procedure.

links at any allocation step in order to keep the size of the allocation array manageable. During allocation array creation, for each allocation we check for signal compatibility (5-V CMOS–3.3-V CMOS, CMOS-TTL, etc.), and add voltage translation buffers.[1] We exclude those allocations for which the pin count, gate count, communication port count, memory limits, and peak power dissipation are exceeded. The allocations in the array are ordered based on dollar cost. If power is being optimized, the ordering is done based on average power dissipation. Once the allocation array is formed, we mark all allocations as unvisited. We pick the unvisited allocation with the least dollar cost, mark it visited, and go through scheduling, performance estimation, and allocation evaluation steps described next.

D. Scheduling

We use a priority-level-based static scheduler for scheduling tasks and edges on all PE's and links in the architecture. This is based on the list scheduling philosophy [16]. We generally schedule only the first copy of the task. The start and finish times of the remaining copies are updated in the association array, as discussed earlier. The remaining copies need to be scheduled only when a required execution slot for a subsequent copy is already occupied by a copy of a previously scheduled higher priority task. This occurs when the higher priority task has a different period or different execution time or different start time. To illustrate this scenario, consider the task graphs shown in Fig. 10(a). For simplicity, assume that there is only one task in each task graph. The numbers next to tasks in Fig. 10(a) denote their execution time on the sole PE-type available. The hyperperiod of these two task graphs is 60. Therefore, three copies of task graph 1 and two copies of task graph 2 are required in the hyperperiod, denoted as 1^1, 1^2, 1^3, and 2^1, 2^2, respectively. The priority levels of tasks 1^1 and 2^1 are -6 and -20, respectively. Hence, task 1^1 is scheduled first. The schedule of tasks 1^2 and 1^3 are derived using the association array by simply adding its period to the schedule

[1] M. McClear, "Low-cost, low-power level shifting in mixed-voltage systems," Applicat. Notes, SCBA002, Texas Instruments, Dallas, TX, 1996.

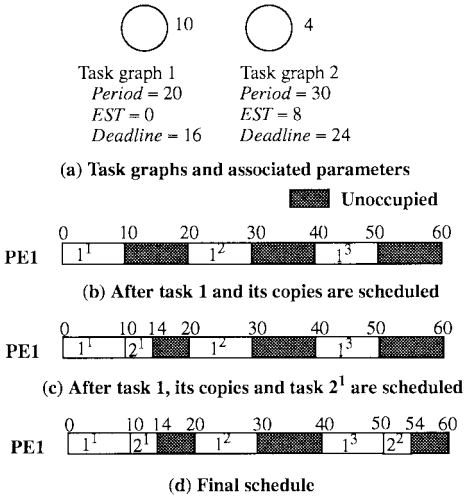


Fig. 10. Scheduling of different copies.

```
SCHEDULE(current_architecture, task graphs,
association array){
    assign deadline-based priority levels to tasks
    and edges;
    initialize the association array;
    for each task t_i which does not start at EST = 0{
        identify the jth copy of t_i, t_i^j, which requires
        a hyperperiod spill;
        t_i^j_priority = t_i^j_priority + hyperperiod;}
    task_list ← order tasks in the order of decreasing
    priority levels;
    for each t_k of task_list{
        t_k_tag = unscheduled;}
    for each unscheduled task t_l from task_list{
        schedule all incoming edges of t_l; //schedules
        //fan-in tasks, if they are not already scheduled
        schedule first copy of t_l and the required
        hyperperiod spill;
        update association array for the remaining copies
        and, if necessary, schedule remaining copies;
        if (t_l is not schedulable){
            return failure;
            break;//reject the current architecture;}
    t_l_tag = scheduled;}
    return success;}
```

Fig. 11. The procedure for scheduling task graphs.

of task 1^1. The resulting schedule is shown in Fig. 10(b). Next, task 2^1 is scheduled in the first available slot {10,14}. The resulting schedule is shown in Fig. 10(c). Based on the schedule of the first copy, the desired schedule for task 2^2 is $\{10+30, 14+30\} = \{40, 44\}$. However, this execution slot is not available. Hence, task 2^2 is scheduled in the first available slot after that, which is {50,54}. The resulting schedule is shown in Fig. 10(d).

We use the procedure outlined in Fig. 11 to schedule tasks and edges. We first identify the copies of tasks which require a hyperperiod spill, and add the hyperperiod to their priority levels. We order 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 processor. Though preemptive scheduling is sometimes not desirable due to the overhead associated with it, it may be necessary to obtain an efficient architecture. In order to decide whether to preempt or not, we use the following criteria. Let ϕ_i and ϕ_j be the priority levels of tasks t_i and t_j, respectively, and let α_{ir} and α_{jr} be their execution times on PE r. Let η_r be the preemption overhead on PE r to which tasks t_i and t_j are allocated. Let $\pi^b(t_i)$ be the best-case finish time (this takes α_{ir} into account) and $\mu(t_i)$ be the deadline of task t_i. We allow preemption of task t_i by t_j under the following two scenarios: 1) $\phi_j > \phi_i$ or 2) t_i is a sink task, and $\pi^b(t_i) + \eta_r + \alpha_{jr} \leq \mu(t_i)$.

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. This is important since scheduling is in the inner loop of co-synthesis. Architectural hints are checked for each task before allowing preemption since an embedded-system specification may require that some critical tasks not be preempted irrespective of their priority levels.

E. System Performance Estimation

We store the best- and worst-case start and finish times of each task and edge. Each node (communication edge) in the task graph has the minimum and maximum entries in the corresponding execution (communication) vector associated with it. When a task (edge) gets allocated, its minimum and maximum execution (communication) times become equal and correspond to the execution (communication) time on the PE (link) to which it is allocated, as shown in Fig. 1(d) (here, cluster *C1* is assumed to be mapped to *PE2*). The FTE step, after each scheduling step, updates the best- and worst-case finish times of all tasks and edges. This is done as follows. Let π^b and π^w represent best- and worst-case finish times, respectively. The best- and worst-case finish times for a task and edge are estimated using the following equations:

$$\pi^b(t_i) = \max\{\pi^b(e) + \alpha^{\min}(t_i)\}$$

and

$$\pi^w(t_i) = \max\{\pi^w(e) + \alpha^{\max}(t_i)\}$$

where $e \in \{E\}$, the set of input edges of t_i

$$\pi^b(e_j) = \pi^b(t_k) + \beta^{\min}(e_j)$$

and

$$\pi^w(e_j) = \pi^w(t_k) + \beta^{\max}(e_j)$$

where t_k is the source node of edge e_j.

Let us next apply the above FTE method to our task graph of Fig. 1(a). Suppose cluster *C1* is allocated to *PE2*, as mentioned before. Then we would obtain the FTE graph of Fig. 1(d), which indicates that the best- and worst-case finish times of sink task $t11$ are 150 and 200, respectively.

F. Allocation Evaluation

Each allocation is evaluated based on the total dollar cost. For hardware cost estimation, we use the incremental cost-estimation approach [17]. We pick the allocation which at

least meets the deadlines in the best case. If no such allocation exists, we pick an allocation for which the summation of the best-case finish times of all task graphs is maximum. The best-case finish time of a task graph is the maximum of the best-case finish times of the constituent tasks with specified deadlines. This generally leads to a less expensive architecture since a larger finish time generally corresponds to a less expensive architecture. If there is more than one allocation which meet this criterion, we choose the allocation for which the summation of the worst-case finish times of all task graphs is maximum. The reason behind using the "maximum" instead of "minimum" in the above cases is that, at intermediate steps, we would like to be as frugal as possible with respect to the total dollar cost of the architecture. If deadlines are not met, we can always upgrade the architecture at a later step.

G. Support of Multiple Supply Voltages

Our co-synthesis algorithm supports a resource library in which different PE's require different supply voltages. This allows mixing of different technologies and derivation of power-efficient architectures by taking advantage of state-of-the-art low-power technology. Support of multiple supply voltages requires checking of signal voltage level compatibility for each communication link/PE interface, inclusion of voltage level translation buffers in the architecture, and estimation of power requirements for multiple voltage levels. The power dissipation in translation buffers is computed considering its average quiescent power dissipation, frequency of the communicating link, and the activity factor of the signal.[1]

Once the architecture is defined, we determine the power-distribution architecture of the system and add the required power-supply converters [18]. This defines the power-supply capacity and the interconnection of various power converters to meet the power requirements of the embedded system.

H. Application of the Co-Synthesis Algorithm

We next apply our co-synthesis algorithm to the task graph of Fig. 1(a). The three clusters shown in Fig. 1(d) are ordered based on the decreasing value of their priority levels. Fig. 12 illustrates the allocation of various clusters. Since cluster *C1* has the highest priority level, it is allocated first to the cheaper processor *PE2* [Fig. 12(a)]. The PFT of the task graph is estimated to be {150, 200} [Fig. 1(d)]. Since the best-case estimated finish time does not meet the deadline, the partial architecture is upgraded. Therefore, *C1* is allocated to processor *PE1* [Fig. 12(b)]. Since the deadline is still not met and all possible allocations are explored, cluster *C1* is marked as allocated and cluster *C2* is considered for allocation. First, an attempt is made to allocate cluster *C2* to the current PE [Fig. 12(c)]. Finish-time estimation indicates that the deadline can be met in the best case. Hence, cluster *C3* is considered for allocation next. Again, an attempt is first made to allocate cluster *C3* to *PE1* [Fig. 12(d)]. Since the deadline is not met in the best case, the architecture needs to be upgraded [Fig. 12(e)]. Since the deadline is still not met, the architecture is upgraded again [Fig. 12(f)]. Now that the deadline is met and all clusters are allocated, the final architecture is given in Fig. 12(f).

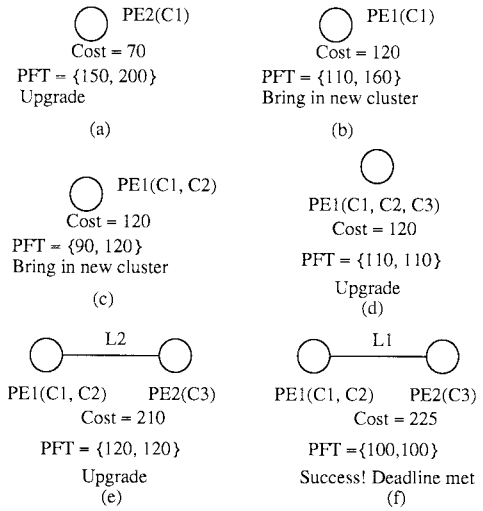


Fig. 12. Stepping through co-synthesis.

IV. Co-Synthesis of Low-Power Embedded Systems

In this section, we describe the co-synthesis system for low-power, called COSYN-LP. The basic co-synthesis procedure outlined in Fig. 2 is also used in COSYN- LP. The parsing and association array formation steps remain the same as before. We describe next how the other steps are modified.

A. Task Clustering

We use deadline-based priority levels to choose a task for clustering. However, once the task is picked, it is grouped with a task along which it has the highest energy level to form clusters. This makes the communication time as well as communication energy for such inter-task edges zero. The energy level concept also takes into account the quiescent energy dissipation in PE's and links. This is why we target energy levels even though our ultimate goal is to minimize average power dissipation subject to the given real-time and peak power constraints.

Energy levels are assigned as follows.

1) For each task t_i (edge e_j), determine the average energy dissipation, as $\alpha^{\max}(t_i)$ ($\beta^{\max}(e_j)$) multiplied by the average power dissipation on the corresponding PE (link). $\alpha^{\max}(t_i)$ and $\beta^{\max}(e_j)$ are chosen because meeting real-time constraints is most important. Mark all tasks as unvisited.
2) For each unvisited task t_i in the task graph, do the following.
   a) If t_i is a sink task, energy level, (t_i) = [average energy of task t_i]. Mark t_i visited.
   b) If t_i is not a sink task, for each edge $e = (t_i, t_f)$ in the set of fan-out edges of task t_i, energy level (t_i) = max (energy level (t_f) + average energy (t_i, t_f) + average energy (t_i)). Mark t_i as visited.

The cluster formation procedure is the same as before, except that we use energy levels instead of priority levels.

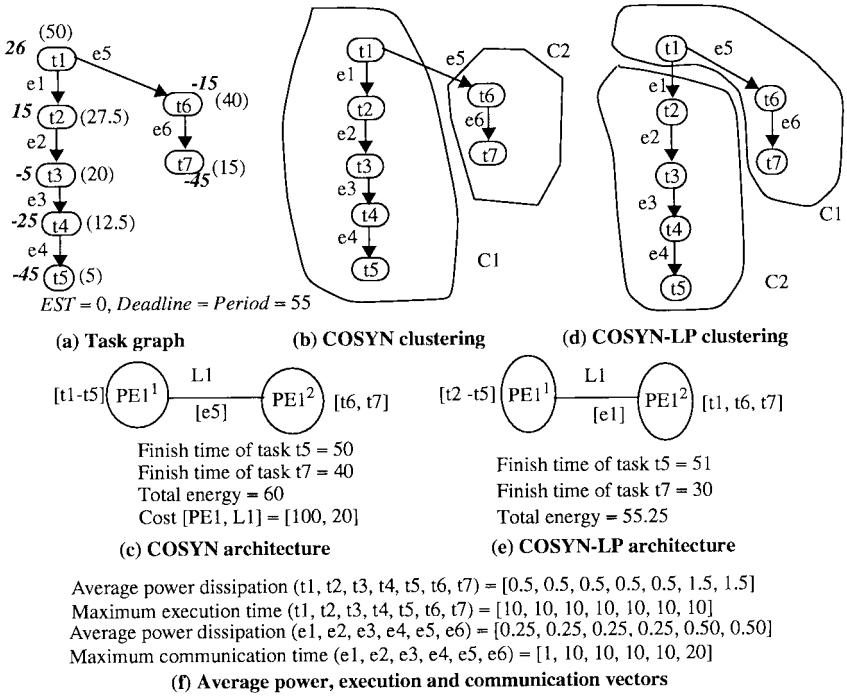


Fig. 13. Task clustering for low power.

The energy levels are recomputed after the clustering of each node. Once clustering is done, we replicate some tasks in more than one cluster to eliminate inter-cluster communication bottlenecks as before [15]. Consider the task graph shown in Fig. 13(a). The numbers in brackets (bold) indicate initial energy (priority) levels. They have been derived from the vectors given in Fig. 13(f). Application of COSYN results in two clusters *C1* and *C2* [Fig. 13(b)], and the architecture shown in Fig. 13(c). For simplicity, only one PE and link are assumed to be present, whose costs are shown in Fig. 13(c). COSYN-LP results in a different clustering [Fig. 13(d)], and the architecture shown in Fig. 13(e). It reduces energy consumption from 60 to 55.25 units with a minor increase in the finish time, while still meeting the deadline. For simplicity, we have assumed the quiescent power dissipation in the PE's/links to be zero. However, in general, we take this into account, as explained later.

B. Cluster Allocation and Performance Evaluation

In the outer loop of co-synthesis, the allocation array is created, as before, for each cluster, and each allocation is checked to see if the peak power dissipation and memory capacity (for general-purpose processors) of the associated PE/link is exceeded. To each link of the allocation, we add the required voltage translation buffer if needed. Entries in the allocation array are ordered based on increasing average power dissipation. If there is more than one allocation with equal average power dissipation, then the allocation with the least dollar cost is chosen. Further ties are broken based on peak power dissipation. In the inner loop of co-synthesis, in addition to FTE, architecture energy/power estimation is also performed as follows.

1) Processor/Link: The average and peak power are estimated based on the tasks (edges) allocated to the processor (link). Let $t_i \in \{T\}$ ($e_j \in \{E\}$) be the set of tasks (edges) assigned to the pth processor P (lth link L). The peak power for P and L are $\max\{\text{peak power } (t_i \in \{T\})\}$ and $\max\{\text{peak power } (e_j \in \{E\})\}$, respectively. These should not exceed the specified peak power constraints. Let $\mathcal{R}^{\xi}$ and θ^{ξ} represent the average energy, and quiescent average power dissipation, respectively. Let Ψ represent the idle time in the hyperperiod. Let n_i (n_j) be the number of times that task t_i (edge e_j) is executed in the hyperperiod. The average energy for P and L is estimated using the following equations:

$$\mathcal{R}^{\xi}(P) = \left[\sum_{t_i \in T} \xi_{ip} \cdot \alpha_{ip} \cdot n_i\right] + [\theta^{\xi}(P) \cdot \Psi(P)]$$

$$\mathcal{R}^{\xi}(L) = \left[\sum_{e_j \in E} \xi_{jl} \cdot \beta_{jl} \cdot n_j\right] + [\theta^{\xi}(L) \cdot \Psi(L)].$$

The average power dissipation of P and L is estimated by dividing their energy dissipation by the hyperperiod.

2) FPGA/ASIC: Tasks assigned to FPGA's and ASIC's can run simultaneously. Therefore, the peak power of an FPGA/ASIC is the summation of the peak power required by all tasks assigned to them and the quiescent power of the unused portion of the FPGA/ASIC. The average energy/power estimation procedure is similar to the one given above.

3) System Power Dissipation: The average power dissipation of the partial architecture is estimated by dividing the total estimated energy in its PE's/links by the hyperperiod.

During the allocation evaluation step, 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-case finish times of the nodes with specified deadlines in all task graphs is maximum.

TABLE I
EXPERIMENTAL RESULTS FOR TASK GRAPHS FROM THE LITERATURE

Example/no. of tasks	Number of PEs/links			Cost ($)			CPU time (sec)		
	Prakash & Parker	Yen/ Hou & Wolf	COSYN	Prakash & Parker	Yen/ Hou & Wolf	COSYN	Prakash & Parker on Solbourne 5/e/900	Yen/Hou & Wolf on Sparc 20	COSYN on Sparc 20
Prakash & Parker (0)/4	1/0	-	1/0	5	-	5	37.0	-	0.20
Prakash & Parker (1)/9	1/0	1/0	1/0	5	5	5	3691.2	59.2	0.40
Prakash & Parker (2)/9	2/1	3/1	2/1	10	10	10	7.4 hrs	56.8	0.54
Prakash & Parker (3)/9	-	3/1	2/1	-	12	10	-	193.3	0.58
Prakash & Parker (4)/13	1/1	-	1/1	5	-	5	106.7hrs	-	0.84
Yen & Wolf Ex/6	-	3/2	3/2	-	1765	1765	-	10.6	0.74
Hou & Wolf Ex1/20	-	2/1	2/1	-	170	170	-	14.9	5.10
Hou & Wolf Ex2/20	-	2/1	2/1	-	170	170	-	5.0	2.64
DSP/119	-	-	2/1	-	-	100	-	-	127.30

V. EXPERIMENTAL RESULTS

Our co-synthesis algorithms, COSYN and COSYN-LP, are implemented in C++. Table I provides results on examples from the literature. Prakash & Parker(0-4) are from [7]. Prakash & Parker(0) is the same as task 1 in [7]. Prakash & Parker(1-3) are the same as task 2 in [7] with different constraints. Prakash & Parker(4) is a combination of task 1 and task 2 from [7]. Yen & Wolf Ex is from [9]. Hou & Wolf Ex(1,2) are from [10]. DSP is from [15] with deadline and period assumed to be 6.5 s. The PE and link libraries used in these results are the same as those used in the corresponding references. As shown in Table I, COSYN consistently outperforms both MILP [7] and iterative improvement techniques [9], [10]. For example, for Prakash & Parker(4), the MILP technique required approximately 107 h of CPU time on Solbourne5/e/900, and Yen and Wolf's algorithm was unable to find a solution, whereas COSYN was able to find the same optimal solution as MILP in less than 1 s on Sparcstation 20 with 256-Mbytes random access memory (RAM).

We also ran COSYN and COSYN-LP on large Bell Laboratories telecom transport systems task graphs representing real-life field applications. They contain tasks for synchronous optical network interface processing, asynchronous transfer-mode cell processing, digital signal processing, provisioning, transmission interfaces, performance monitoring, protection switching, etc. Thy have wide variations in their periods ranging from 25 μs to 1 min. On an average, over 70% of tasks in the task graphs are of different types. The general-purpose processors 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 256-Kbyte second-level cache), 11 ASIC's, one XILINX 3195A FPGA, and one ORCA 2T15 FPGA. For each general-purpose processor, four DRAM banks providing up to 64-Mbyte capacity were evaluated. DRAM devices with 60-ns access time were used. The ASIC's were based on the existing designs of various telecommunication systems. For new functions, macro blocks were synthesized for various standard-cell technologies and FPGA families. The link library was assumed to contain a 680X0 bus, a 10-Mb/s LAN, and a 31-Mb/s serial link.

TABLE II
COSYN WITH ASSOCIATION ARRAY AND CLUSTERING

Example/ no. of tasks	No. of PEs/ links	Cost($)	CPU time (sec)	Average power dissipation (Watts)
BETS1/15	2/1	305	0.54	4.43
BETS2/45	4/3	455	1.42	7.72
BETS3/156	13/11	1725	118.40	26.40
BCS/318	18/7	19800	910.50	198.40
ATMIF/512	24/7	11800	1419.40	214.60
BATIF1/728	28/11	14214	3942.70	238.24
BATIF2/845	35/12	16088	10418.64	307.10
OASIF/1072	44/16	27145	16864.70	381.70

TABLE III
COSYN WITHOUT ASSOCIATION ARRAY

Example/ no. of tasks	No. of PEs/ links	Cost($)	CPU time (sec)	Average power dissipation (Watts)
BETS1/15	2/1	305	1.61	4.18
BETS2/45	4/3	455	3.46	7.10
BETS3/156	13/11	1725	1515.32	25.85
BCS/318	16/8	19550	8829.91	189.80
ATMIF/512	22/7	11610	10503.56	195.40
BATIF1/728	26/11	14080	33221.19	232.60
BATIF2/845	32/11	15800	72557.46	299.85
OASIF/1072	43/15	26895	78248.30	364.30

Results in Tables II–VI show that COSYN was also able to handle the large telecom transport system task graphs efficiently. Note that even architectures with the same number of PE's and links can have different cost because of different PEs/links that may have been used. Also, two architectures with equal cost and the same number and type of PE's and link can still have different power dissipation since they may employ different schedules with different number of

TABLE IV
COSYN WITHOUT CLUSTERING

Example/ no. of tasks	No. of PEs/ links	Cost($)	CPU time (sec)	Average power dissipation (Watts)
BETS1/15	2/1	305	2.81	4.25
BETS2/45	4/3	455	3.57	7.45
BETS3/156	13/11	1725	374.60	24.95
BCS/318	17/8	19650	1425.41	191.32
ATMIF/512	23/7	11700	3104.38	208.95
BATIF1/728	27/11	14100	12380.10	234.46
BATIF2/845	33/12	16005	18961.53	301.45
OASIF/1072	42/14	25995	38204.75	370.52

TABLE V
COSYN WITHOUT ASSOCIATION ARRAY AND CLUSTERING

Example/ no. of tasks	No. of PEs/ links	Cost ($)	CPU time (sec)	Average power dissipation (Watts)
BETS1/15	2/1	305	4.49	4.11
BETS2/45	4/3	455	6.87	7.10
BETS3/156	13/11	1725	1749.95	24.35
BCS/318	16/8	19550	10088.34	187.60
ATMIF/512	22/7	11590	14364.33	193.82
BATIF1/728	26/11	14005	44434.28	230.20
BATIF2/845	32/11	15750	90537.98	294.62
OASIF/1072	42/14	25890	120412.96	363.11

TABLE VI
COSYN USING A RESOURCE LIBRARY WITH 5-V PE'S ONLY

Example/ no. of tasks	No. of PEs/links	Cost($)	CPU time (sec)	Average power dissipation (Watts)
BETS1/15	4/2	410	0.45	8.93
BETS2/45	9/4	612	1.67	12.46
BETS3/156	Architecture is not feasible			
BCS/318	Architecture is not feasible			
ATMIF/512	25/5	12400	1400.10	298.72
BATIF1/728	30/12	15308	3890.51	356.60
BATIF2/845	37/14	17145	10335.60	411.62
OASIF/1072	47/17	27850	15989.38	509.56

preemptions. For the results in Table II, COSYN was allowed to use both the association array and task clustering. In Table III, it was allowed the use of task clustering, but not association array. In Table IV, it was allowed the use of association array, but not task clustering. Finally, in Table V, it was not allowed the use of either association array or task clustering. From Tables II and III, we can see that the association array concept reduces CPU time by an average of 81% (average is based on individual reductions) at an average increase of 0.8% in embedded-system cost. From Tables II and IV, we can see that task clustering reduces CPU time by an average of 59% at an average increase of 0.9% in embedded-system cost. From Tables II and V, we can see that the combination of the association array concept and task clustering results in an average reduction of 88% in CPU time at an average increase of 1.4% in embedded-system cost. This enables the application of COSYN to very large task graphs. However, since CPU time is not a big concern for smaller task graphs with a well-behaved hyperperiod, we have provided flags in our co-synthesis system to allow the user to bypass association array formation or task clustering or both. Tables II and VI show the importance of using a resource library, which includes PE's operating at different supply voltages. While using a resource library with only 5-V PE's, the architecture was not feasible for BETS3 and BCS since some of the associated tasks required PE's with different supply voltages. Support of multiple supply voltages results in an average reduction of 33.4% in power dissipation and 12% in embedded-system cost.

TABLE VII
COSYN-LP

Example/ no. of tasks	No. of PEs/ links	Cost ($)	CPU time (sec)	Average power dissipation (Watts)	Measured Average power dissipation (Watts)
BETS1/15	3/1	368	1.20	2.66	2.45
BETS2/45	5/4	554	2.20	5.73	5.29
BETS3/156	12/10	1993	142.56	23.57	22.18
BCS/318	16/8	22400	1018.72	161.34	154.10
ATMIF/512	25/9	12760	1609.63	178.15	166.90
BATIF1/728	28/12	16224	5046.70	211.15	199.80
BATIF2/845	35/12	17600	12166.84	240.80	225.60
OASIF/1072	47/18	29425	19910.21	337.42	312.41

Table VII gives the results for COSYN-LP. COSYN-LP was able to reduce power dissipation by an average of 19.6% over COSYN (Table II) at an average increase of 13.9% in embedded-system cost. For these results, both COSYN and COSYN-LP were supplied with a resource library with PE's operating at different supply voltages. Also, as shown in the last column of Table VII, the actual system power measurements made on the COSYN-LP architectures indicate that the error of the COSYN-LP power estimator is within 8%.

VI. CONCLUSION

We presented an efficient distributed system co-synthesis algorithm. Even though it is a heuristic, experimental results show that it produces optimal results for the examples from the literature. It provides several orders of magnitude advantage in CPU time over existing algorithms. This enables its application to large examples for which experimental results are very encouraging. Large real-life examples have not been tackled previously in the literature. We have also presented one of the first co-synthesis algorithms for power optimization. COSYN is currently being used in Lucent Bell Laboratories to tackle the next generation telecom transport system task graphs.

REFERENCES

[1] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. San Francisco, CA: Freeman, 1979.
[2] A. Kalavade and P. A. Subrahmanyam, "Hardware/software partitioning for multi-function systems," in *Proc. Int. Conf. Computer-Aided Design*, Nov. 1997, pp. 516–521.

[3] F. Balarin *et al.*, *Hardware-Software Co-Design of Embedded Systems: The POLIS Approach.* Norwell, MA: Kluwer, 1997.
[4] H. De Man *et al.*, "Hardware–software codesign of digital telecommunication systems," *Proc. IEEE*, vol. 85, Apr. 1997.
[5] D. D. Gajski, F. Vahid, S. Narayan, and J. Gong, "SpecSyn: An environment supporting the specify-explore-refine paradigm for hardware/software system design," Univ. California, Riverside, CA, Tech. Rep. CS-96-08, Sept. 1996.
[6] W. Wolf, "Hardware-software co-design of embedded systems," *Proc. IEEE*, vol. 82, pp. 967–989, July 1994.
[7] S. Prakash and A. Parker, "SOS: Synthesis of application-specific heterogeneous multiprocessor systems," *J. Parallel & Distrib. 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. Workshop Hardware–Software Co-Design*, Sept. 1994, pp. 34–41.
[9] T.-Y. Yen and W. Wolf, "Communication synthesis for distributed embedded systems," in *Proc. Int. Conf. Computer-Aided Design*, Nov. 1995, pp. 288–294.
[10] J. Hou and W. Wolf, "Process partitioning for distributed embedded systems," in *Proc. Int. Workship Hardware–Software Co-Design*, Sept. 1996, pp. 70–76.
[11] D. Kirovski and M. Potkonjak, "System-level synthesis of low-power real-time systems," in *Proc. Design Automation Conf.*, June 1997, pp. 697–702.
[12] S. Srinivasan and N. K. Jha, "Hardware-software co-synthesis of fault-tolerant real-time distributed embedded systems," in *Proc. European Design Automation Conf.*, Sept. 1995, pp. 334–339.
[13] Y.-T. S. Li, S. Malik, and A. Wolfe, "CINDERELLA: A retragetable environment for performance analysis of real-time software," in *Proc. Euro-Par.*, 1997.
[14] E. Lawler and C. Martel, "Scheduling periodically occurring tasks on multiple processors," *Inf. Process. Lett.*, vol. 7, pp. 9–12, Feb. 1981.
[15] S. Yajnik, S. Srinivasan, and N. K. Jha, "TBFT: A task based fault tolerance scheme for distributed systems," in *Proc. ISCA Int. Conf. Parallel & Distrib. Comput. Syst.*, Oct. 1994, pp. 483–489.
[16] K. Ramamritham and J. A. Stankovic, "Scheduling algorithms and operating systems support for real-time systems," *Proc. IEEE*, vol. 82, pp. 55–67, Jan. 1994.
[17] F. Vahid and D. Gajski, "Incremental hardware estimation during hardware/software functional partitioning," *IEEE Trans. VLSI Syst.*, vol. 3, pp. 459–464, Sept. 1995.
[18] B. P. Dave, "Hardware/software co-design of heterogeneous real-time distributed embedded systems," Ph.D. dissertation, Elect. Eng. Dept., Princeton University, Princeton, NJ, 1998.

Bharat P. Dave (M'91) received the B.Tech. degree in electronics and communications in India, the M.S. degree in electrical engineering from Virginia Polytechnic Institute and State University, Blacksburg, VA, and the M.A. and Ph.D. degrees in electrical engineering from Princeton University, Princeton, NJ.

Since 1986, he has been with Bell Laboratories, Lucent Technologies, Holmdel, NJ, where he is currently a Distinguished Member of Technical Staff. His research interests include hardware/software co-design, fault-tolerant computing, optical networks, distributed and real-time systems, telecom systems, and mobile computing.

Dr. Dave received the William C. Carter Award for his paper at the IEEE International Symposium on Fault-Tolerant Computing in 1997. He also co-authored a paper which was nominated for the Best Paper Award at the DATE98 Conference.

Ganesh Lakshminarayana, for photograph and biography, see this issue, p. 14.

Niraj K. Jha (S'85–M'85–SM'93–F'98), for photograph and biography, see this issue, p. 4.