COSYN: Hardware-Software Co-Synthesis of Embedded Systems*

Bharat P. Dave[1], Ganesh Lakshminarayana, and Niraj K. Jha
Department of Electrical Engineering
Princeton University, Princeton, NJ 08544

Abstract: Hardware-software co-synthesis is the process of partitioning an embedded system specification into hardware and software modules to meet performance, power and cost goals. In this paper, we present a co-synthesis algorithm which starts with periodic task graphs with real-time constraints and produces a low-cost heterogeneous distributed embedded system architecture meeting the constraints. The algorithm has the following features: 1) it allows the use of multiple types of processing elements (PEs) and inter-PE communication links, where the links can take various forms (point-to-point, bus, local area network (LAN), etc.), 2) it supports both concurrent and sequential modes of communication and computation, 3) it allows both preemptive and non-preemptive scheduling, 4) it employs the concept of an association array to tackle the problem of multi-rate systems (which are commonly found in multimedia applications), 5) it uses a scheduler based on dynamic deadline-based priority levels for accurate performance estimation of a co-synthesis solution, 6) it uses a new task clustering technique which takes the dynamic nature of the critical path, and the existence of multiple critical paths in the task graph into account, and 7) if desired, it also optimizes the architecture for power consumption (we are not aware of any other co-synthesis algorithm that optimizes power). Application of the proposed algorithm to examples from the literature and real-life telecom transport systems shows its efficacy.

1 Introduction

The architecture of embedded systems is generally defined based on the experience of system architects, and at times, it is either over-designed or fails to meet the requirements. 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. Both allocation and scheduling are known to be NP-complete [1]. Therefore, optimal co-synthesis is computationally a very hard problem. In addition, since many embedded systems are used in mobile applications, both peak and average power consumption have become important concerns. The peak power consumption determines the packaging cost and the average power consumption determines the battery life. Thus, it is also important to optimize power consumption during co-synthesis.

Previous researchers have mostly focused their interest on hardware-software co-synthesis of one-CPU-one-ASIC architectures [2,3]. However, distributed embedded system architectures can employ multiple CPUs, ASICs, and field-programmable gate arrays (FPGAs). Two distinct approaches have been used for distributed system co-synthesis: optimal and heuristic. In the optimal domain, the two approaches are mixed integer linear programming (MILP) and exhaustive. The MILP solution [4] has the following limitations: 1) it is restricted to one task graph, 2) it does not handle preemptive scheduling, 3) it requires determination of the interconnection topology upfront, and 4) because of time complexity, it is suitable only for small task graphs. A configuration-level hardware-software partitioning algorithm is presented in [5] based on an exhaustive enumeration of all possible solutions, which is also impractical for large task graphs. There are two distinct approaches in the heuristic domain: iterative and constructive. In [6,7], an iterative procedure is given. It considers only one type of communication link and does not allow mapping of each successive instance of a periodic task to different PEs. A constructive co-synthesis procedure for fault-tolerant distributed embedded systems is proposed in [8]. However, it does not support communication topologies such as bus, LAN *etc*., and it uses a pessimistic performance estimation technique. It is also not suitable for multi-rate embedded systems, *e.g.* multi-media systems. Also, power consumption has not been optimized in any of these co-synthesis techniques.

We have developed a heuristic-based co-synthesis technique, called COSYN, which includes the allocation, scheduling and performance estimation steps as well as power optimization features. Our technique is suited for 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 CPU time, quality of solution and number of features. In fact, for the task graphs from the literature for which MILP-based optimal results are known, COSYN also obtained the same optimal results in many orders of magnitude smaller CPU time. However, in general, of course, being a heuristic, COSYN cannot guarantee optimality.

2 Definitions and Basic Concepts

Embedded systems consist of off-the-shelf general-purpose processors, ASICs and FPGAs and perform application-specific functions. The *hardware architecture* of an embedded system defines the type and interconnection of various hardware modules. Its *software architecture* defines the allocation of sequence of codes to specific general-purpose processors. The embedded system functionality is usually described through a set of acyclic *task graphs*, whose nodes represent tasks. Tasks communicate data to each other, indicated by a directed edge between two communicating tasks. We focus on periodic task graphs. Each such graph has an earliest start time (*est*), period, and deadline, as shown for an example in Figure 1(a). Each task of a periodic task graph inherits the graph's period and can have a different deadline. The deadline of the task graph is the maximum of all such deadlines.

We define execution_vector $(t_i) = \{\alpha_{i1}, \alpha_{i2},.., \alpha_{in}\}$ to be an *execution vector* of task t_i, where α_{ij} indicates the execution time of t_i on PE j from the PE library. The preference_vector $(t_i) = \{\gamma_{i1}, \gamma_{i2},.., \gamma_{in}\}$ is a *preference vector* of task t_i, where γ_{ij} indicates preferential mapping for t_i. If γ_{ij} is 0, it indicates that t_i cannot be executed on PE j, and 1 if there are no constraints. This vector is useful in cases where preferred allocation is determined based on prior experience or task characteristics. Similarly, the *exclusion vector* of task t_i, exclusion_vector $(t_i) = \{\delta_{i1}, \delta_{i2},..., \delta_{iq}\}$, specifies whether certain tasks can co-exist on the same PE, *i.e.* $\delta_{ij} = 1$ 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. 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. The preference vector of a cluster indicates which PEs the cluster cannot be allocated to. Similarly, we define exclusion_vector(C_i) of cluster C_i as the bit-wise logical OR of the exclusion vectors of

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

[1] also at Bell Laboratories, Lucent Technologies, 101 Crawfords Corner Road, Holmdel, NJ 07733.

DAC 97, Anaheim, California.

all the tasks in the cluster. A task t_i 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_i does not result in the 0-vector, *i.e.* a vector with all elements 0. If all elements of a preference vector of cluster C_i are 0, it 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 0. This indicates that tasks in cluster C_i can be co-allocated with task t_j.

We define communication_vector(e_k) = (β_{k1}, β_{k2},..., β_{km}} to be the communication vector of task graph edge e_k, where β_{kl} indicates the time it takes to communicate data on e_k on communication link l from the link library. The communication vector for each edge 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. The access_time_vector(l) = [Ω_{l1}, Ω_{l2}, ..., Ω_{lm}] is an access time vector for link l, where Ω_{lr} represents the access time per packet with r number of ports on l. Suppose l has s ports. Let τ_l be the communication time of a packet on l. Then β_{kl} is given by

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

At the beginning of co-synthesis, since the actual number of ports on the links is not known we initially use an average number of ports (specified *a priori*) to determine the communication vector.

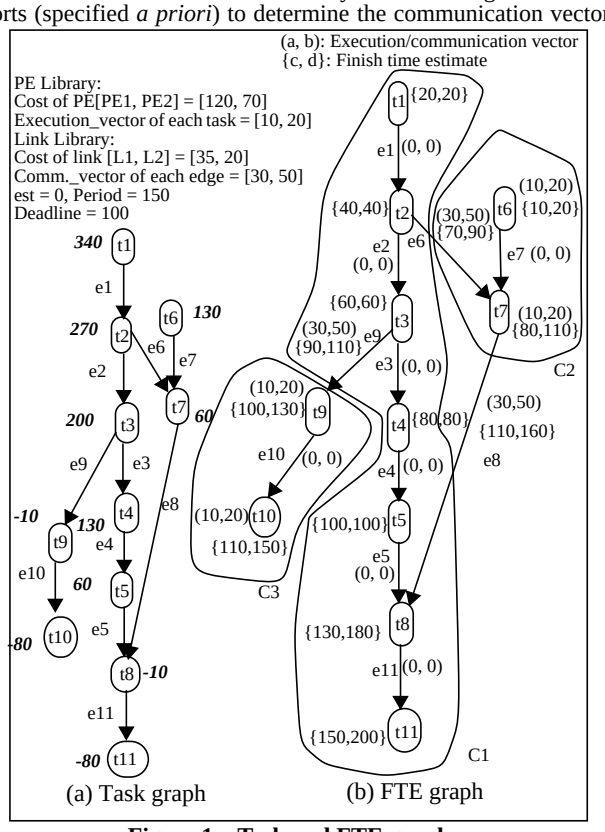

Figure 1. Task and FTE graphs

This vector is recomputed after each allocation, considering the actual number of ports on the link. The average_power_vector (t_i) = {ξ_{i1}, ξ_{i2},.., ξ_{in}} is an average power vector of t_i, where ξ_{ij} indicates the average power consumption of task t_i on PE j. The average power is determined considering normal operating conditions, *e.g.* nominal voltage levels, average data stream *etc*. Similarly, peak_power_vector (t_i) = {κ_{i1}, κ_{i2},.., κ_{in}} is a peak power vector of task t_i, where κ_{ij} indicates the peak power consumption of t_i on PE j. The peak power dissipation is determined considering worst-case operating conditions, *e.g.* worst-case operating voltage, worst-case data stream *etc*. The preference vector, exclusion vector, 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 memory architecture of embedded systems plays an important role from both performance and cost point of view. Previous algorithms have generally ignored this aspect. 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 allocation, we check whether the available memory capacity has been exceeded.

For each available processor, its cost, average/peak quiescent power consumption, and associated peripheral attributes such as memory architecture, processor-link communication characteristics, and cache characteristics are assumed to be specified. For each ASIC, its cost, package attributes such as available pins, available gates, and average and peak power dissipation per gate are assumed to be specified. Similarly, for each FPGA, its cost, average/peak quiescent power, package attributes such as available pins, and the maximum number of flip-flops or combinational logic blocks (CLBs) or programmable functional units (PFUs) are assumed to be specified. Another important attribute of an FPGA is the boot memory requirement which needs to be allocated for the FPGA. Generally, all flip-flops/CLBs/PFUs are not usable due to routing restrictions. We take this into account through a term called the *effective usage factor* (EUF). We allow the user to specify an EUF based on his/her own experience. The user can also specify the *effective pin usage factor* (EPUF) to indicate what percentage of package pins can be used for allocation.

Each communication link is characterized by: 1) the number of information bytes per packet, 2) link access time vector, 3) communication time per packet, and 4) average/peak quiescent power. The PE and link libraries together form the *resource library.*

3 The COSYN Algorithm

In this section, we first provide an overview of COSYN and then follow up with details on each step. Figure 2 presents one possible co-synthesis process flow which we follow in our work. In the parsing step, the task graphs, system/task constraints, and resource library are parsed and appropriate data structures are created. The hyperperiod of the system is computed as the least common multiple (LCM) of the periods of various task graphs. 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 [9]. 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 using the concept of an *association array* introduced later. The *clustering* step involves grouping of tasks to reduce the search space for the allocation step [8, 10]. This significantly reduces the overall complexity of the co-synthesis algorithm since allocation is part of the inner loop of this algorithm. Clusters are ordered based on their importance/priority. The allocation step determines mapping of tasks (edges) to PEs (communication links). There are two loops in this co-synthesis process flow: 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 all possible allocations 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 pinout, gate count, and package power dissipation is not exceeded. 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.

Generally, several tasks are reused across multiple functions and an efficient co-synthesis algorithm should exploit this fact. This can be done through architectural hints. Such a hint for each task indicates whether the task's architecture may be reused. Our algorithm uses these hints for *architectural reuse* based on a previously stored allocation solution for the task to provide a very

efficient co-synthesis platform for large embedded systems. Architectural hints are also used to indicate whether the given task is suitable for preemption or not.

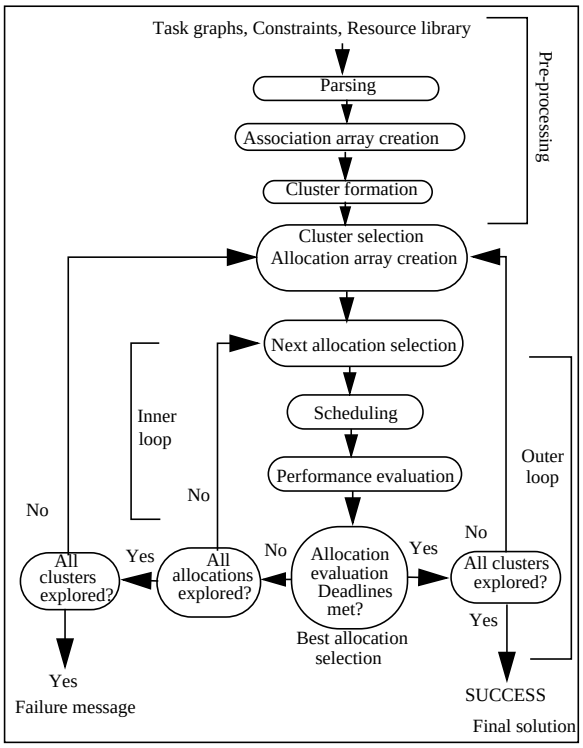

Figure 2. The co-synthesis process flow

The next step is *scheduling* which determines the relative ordering of task/communication execution and the start and finish times for each task and edge. We support both preemptive and non-preemptive static scheduling. We also take into consideration the operating system overheads such as 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*. An important part of performance evaluation is *finish-time estimation* (FTE). This process uses the start and finish times of each task and estimates whether the tasks with specified deadlines meet those deadlines or not. The *allocation evaluation* step compares the current allocation against previous ones based on total dollar cost. If there are more than one allocation with equal dollar cost, we pick the allocation with the lowest average power consumption (assuming power optimization is a secondary objective). Other attributes such as memory requirements and peak power dissipation can also be used to further evaluate the allocations.

3.1 The Association Array

It was shown in [9] that there exists a feasible schedule for a job if and only if there exists a feasible schedule for the hyperperiod. This requires each task graph to be replicated the requisite number of times in the hyperperiod. The advantage of this approach is that it allows different instances of a task to be allocated to different PEs. However, this flexibility comes at a severe price in terms of co-synthesis CPU time and memory requirement when the hyperperiod is large compared to the periods. This could happen, for example, if the periods are comparable, but co-prime, or when one period is much larger than the others. One way to tackle this problem is through the use of an analytical technique such as fixed-point iteration [6,7], which does not require any task graph replication. However, this comes at the price of not allowing different instances of the task to be allocated to different PEs, thus, potentially, increasing system cost. In order to address the limitations of both methods, we propose to employ two approaches: 1) task graph period adjustment to reduce the hyperperiod, and 2) forming an association array. In the first approach, we shorten some of the periods by a small user-adjustable amount (up to 3%) to reduce the hyperperiod. 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 the co-synthesis solution or the cost of the distributed architecture.

We use the concept of an association array to avoid the actual replication of task graphs. An association array has an entry for each task of each copy of the task graph and contains information such as: 1) the PE to which it is allocated, 2) its priority level, 3) its deadline, 4) its best-case finish time, and 5) its worst-case finish time. The deadline of the nth instance of a task graph is offset by $(n-1)$ multiplied by its period from the deadline in the original task graph. The association array not only eliminates the need to replicate the task graphs, but it also allows allocation of different task graph instances to different PEs, if desirable, to derive an efficient architecture. 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. 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.

3.2 Task Clustering

Clustering involves grouping of tasks to reduce the complexity of allocation. Our clustering technique addresses the fact that there may be multiple longest paths through the task graph and the length of the longest path changes after partial clustering. In order to cluster tasks, we first assign deadline-based priority levels to tasks and edges using the following procedure. A non-sink task t_j may either have a deadline or not. We define $\omega(t_j)$ to be equal to the deadline of t_j if the deadline is specified, and ∞ otherwise.

a. Priority level of sink task t_i = execution time (t_i) – deadline (t_i).
b. Priority level of non-sink task t_j = max (priority level of its fanout task t_f + communication time of edge (t_j, t_f), $-\omega(t_j)$) + execution time (t_j).
c. Priority level of edge e_k = priority level of destination node (e_k) + communication time (e_k).

As an example, the numbers adjacent to nodes in Figure 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 as well as the deadline. In order to reduce the schedule length, we need to decrease the length of the longest path which is done by forming a cluster of the tasks along the longest path. This makes 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.

At the beginning, we sort all tasks in the order of decreasing priority levels. We pick 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 already with another fanout task, 2) the fan-in task's cluster C_k is preference- and exclusion-compatible with t_i, and 3) the cumulative size of tasks in C_k does not exceed the cluster size threshold. If the fan-in set of t_i is not empty, we identify an eligible cluster which is grown (*i.e.* expanded) using a cluster growth procedure. 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. In order to ensure load balancing among various PEs of the architecture, the cluster size should be limited. If the cluster size is too big, it may be prevented from being allocated to any PE. If it is too small, it would increase the total number of clusters and increase the computational complexity. We use a parameter called cluster size threshold, C_{th}, to limit the size of the cluster. C_{th} is set equal to the hyperperiod. At any point in the clustering procedure, for any cluster C_k containing m tasks, its size, denoted as θ_k, is estimated by the following equation. Let p_i denote the period of

task t_i of cluster C_k and let Γ be the hyperperiod. Then

$$\theta_k = \sum_{i=1}^{m} (\pi^w(t_i) \cdot (\Gamma \div (p_i)))$$

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 fan-out tasks of t_i along which the priority level of t_i is the highest. We recompute 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 allows us to identify the critical path dynamically and to facilitate compression of multiple critical paths.

The application of the clustering procedure to the task graph of Figure 1(a) results in three clusters, C1, C2, and C3. Once the clusters are formed, some tasks are replicated in two or more clusters to address inter-cluster communication bottlenecks [10].

3.3 Cluster Allocation

We define the priority level of a cluster to be the maximum of the priority levels of the constituent tasks. Clusters are ordered based on decreasing priority levels. We use a dynamic priority level, *i.e.* 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 the allocation array in the order of increasing value of the cost function. Once the allocation array is formed, we use the inner loop of co-synthesis to evaluate the allocations.

3.3.1 The Outer Loop of Co-Synthesis

The allocation array considers the following: 1) architectural hints, 2) preference vector, 3) allocation of the cluster to existing resources in the partial architecture, 4) upgrade of links, 5) upgrade of PEs, 6) addition of PEs, and 7) addition of links. Architectural hints are used to pre-store allocation templates (these templates correspond to the mapping of sub-task-graphs to part of the architecture being built). We exclude those solutions for which the pin count, gate count, and memory limits (and power dissipation if power is being optimized) are exceeded. Once an allocation array is formed, the allocations in it are ordered based on dollar cost. If power is being optimized, the ordering is done based on average power dissipation while making sure the peak power dissipation is within limits.

3.3.2 The Inner Loop of Co-Synthesis

We pick the unvisited allocation with least dollar cost, mark it visited and go through scheduling and solution evaluation steps described next. We use a priority-level based scheduler for scheduling tasks and edges on all PEs and links in the allocation. 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. Usually, this is sufficient to derive an efficient architecture. However, we do sometimes need to schedule the remaining copies [11]. This is followed by the performance estimation step.

3.4 Scheduling

We first order tasks and edges based on the decreasing order of their priority levels. If two tasks (edges) have equal priority levels, 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 a task 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 (PO) 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 scenarios: 1) If $\phi_j > \phi_i$, else 2) If t_i is a sink task, and $\pi^b(t_i) + \eta_r + \alpha_{jr} \leq \mu(t_i)$. η_r is specified *a priori*. It includes context switching and any other processor-specific overheads. 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.

We support task graphs which do not start execution at time t=0. This, in conjunction with a task graph whose period is less than its deadline, may require that some tasks finish after the hyperperiod. We allow the schedule of some tasks to spill over the hyperperiod and ensure that resources are not overused, using the concept of spill vector and deadline re-assignment [11].

3.4.1 Performance Estimation

The scheduler provides an accurate information on the start and finish times of the tasks in the allocated clusters. This, in turn, makes our FTE method more accurate and minimizes the false rejection of an allocation. Each node (communication edge) in the task graph has best- and worst-case execution (communication) times corresponding to the minimum and maximum entries in the corresponding execution (communication) vector. When a task (edge) gets allocated, its best- and worst-case execution (communication) times become equal and corresponds to the execution (communication) time on the PE (link) to which it is allocated. The FTE step, after each scheduling step, updates the best- and worst-case finish times of all tasks. Scheduling after each allocation step greatly improves the FTE accuracy compared to other approaches where FTE assumes worst-case allocation [8], which often results in pessimistic estimates. The best- and worst-case finish times, π^b and π^w respectively, of each task and edge are estimated as follows. Let α_i^b and α_i^w (β_j^b and β_j^w) represent the best- and worst-case execution (communication) times of task t_i (edge e_j), respectively. The best- and worst-case finish times for a task and edge are estimated using the following equations.

1. $\pi^b(t_i) = \max\{\pi^b(e) + \alpha_i^b\}$ and $\pi^w(t_i) = \max\{\pi^w(e) + \alpha_i^w\}$ where $e \in \{E\}$, the set of input edges of t_i.
2. $\pi^b(e_j) = \pi^b(t_k) + \beta_j^b$ and $\pi^w(e_j) = \pi^w(t_k) + \beta_j^w$ where t_k is the source node of edge e_j.

Let us next apply the above FTE method to the task graph in Figure 1(a). Suppose cluster C1 is allocated to PE2, and the other two clusters are unallocated. We would then obtain the FTE graph shown in a Figure 1(b) which indicates that the best- and worst-case finish times of sink task t11 are 150 and 200, respectively.

3.4.2 Allocation Evaluation

Each allocation is evaluated based on the total dollar cost. 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 time of all tasks with specified deadlines in all task graphs is maximum. This may seem counter-intuitive. However, this generally leads to a less expensive architecture since larger finish times generally correspond to less expensive processor/link. If there are more than one allocation which meet this criterion then we choose the allocation for which the summation of the worst-case finish times of all tasks with deadlines 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. Since we allow addition as well as upgrade of PEs/links during co-synthesis, the real-time constraints will ultimately be met.

3.5 Application of the Co-Synthesis Algorithm

We next apply the above co-synthesis algorithm to the task graph in Figure 1(a). Clusters are ordered based on the decreasing value of their priority levels. Figure 3 illustrates the allocation of various clusters during the outer and inner loops of co-synthesis. Since cluster C1 has the highest priority level, it is allocated first to the cheaper processor PE2, as shown in Figure 3(a). The scheduler is run and the projected finish time (PFT) is {150, 200}, as shown in Figure 1(b). Since the best-case estimated finish time does not meet the deadline, the partial architecture needs to be upgraded. Therefore, C1 is allocated to processor PE1, as shown in Figure 3(b). Since deadlines are still not met and all possible allocations are explored, C1 is marked as allocated and cluster C2 is considered for allocation. First, an attempt is made to allocate C2 to the current PE, as shown in Figure 3(c). After scheduling, FTE indicates that

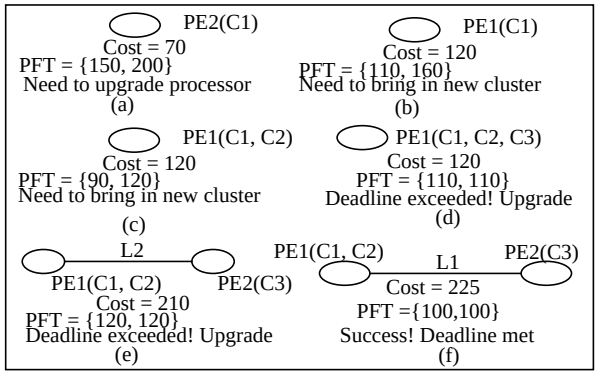

Figure 3. Stepping through COSYN

the deadlines can be met in the best case. Hence, next, C3 is considered for allocation. Again, an attempt is first made to allocate C3 to PE1, as shown in Figure 3(d). Since the deadline is not met in the best case, the architecture needs to be upgraded, as shown in Figure 3(e). Since the deadline is still not met, the architecture is upgraded again, as shown in Figure 3(f). Now that the deadline is met and all clusters are allocated, the architecture given in Figure 3(f) is the final solution.

4 Co-Synthesis of Low Power Embedded Systems

For some embedded systems, along with cost and real-time constraints, another important constraint is power dissipation. Hence, an efficient co-synthesis technique targeted towards synthesis of low power embedded systems is of utmost importance.

The basic co-synthesis process flow of Figure 2 is also used in the co-synthesis system for low power, termed COSYN-LP. The parsing, association array formation and scheduling steps remain the same as before. We describe next how the other steps are modified.

4.1 Clustering

We use deadline-based priority levels to identify the order for clustering tasks, however, we use energy levels instead of priority levels to form clusters, since our objective is to minimize overall power consumption. Clustering along the higher energy-level path makes the communication time as well as communication energy for inter-task edges zero. The concept of energy levels also lets us take into account the quiescent energy dissipation in PEs and links in a straightforward manner. This is the reason we target energy levels even though our ultimate goal is to reduce average power dissipation subject to the given real-time and peak power constraints. Energy levels are assigned as follows:

1. For each task and edge, determine the average energy dissipation, as the worst-case execution or communication time (derived from the execution/communication vector) multiplied by the corresponding average power dissipation (the worst-case execution/communication time is 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: if t_i is a sink task, energy level (t_i) = [average energy of task t_i]. If t_i is not a sink task, for each edge $e = (t_i, t_f)$ in the set of fanout edges of task t_i, where t_f is a fanout task, energy level (t_i) = max (energy level (t_f) + average energy (t_i, t_f) + average energy (t_i)). Mark t_i visited.

The cluster formation procedure is the same as the one discussed in Section 3.2 except that we use energy levels instead of priority levels. The energy levels are recomputed after the clustering of each node. The objective of clustering in this context is to try to decrease the system energy consumption while still meeting real-time constraints. To appreciate the difference between the energy-level based clustering and the one given in Section 3.2, consider the task graph shown in Figure 4(a). The numbers in brackets in this figure indicate energy levels and the numbers in bold indicate priority levels. They have been derived from the vectors given in Figure 4(f). Application of COSYN results in two clusters C1 and C2, as shown in Figure 4(b), and the architecture shown in Figure 4(c). Here, for simplicity, only one PE and link are assumed to be present in the PE and link libraries, whose costs are shown in Figure 4(c). COSYN-LP results in a different clustering, as shown in Figure 4(d). The resulting architecture from COSYN-LP is shown in Figure 4(e). COSYN-LP results in a reduction in overall energy consumption from 60 to 55.25 with a minor increase in the finish time while still meeting the deadline. Here, for simplicity, we have assumed that the quiescent power dissipation in the PEs/links is zero. However, in general, we take this into account, as explained later.

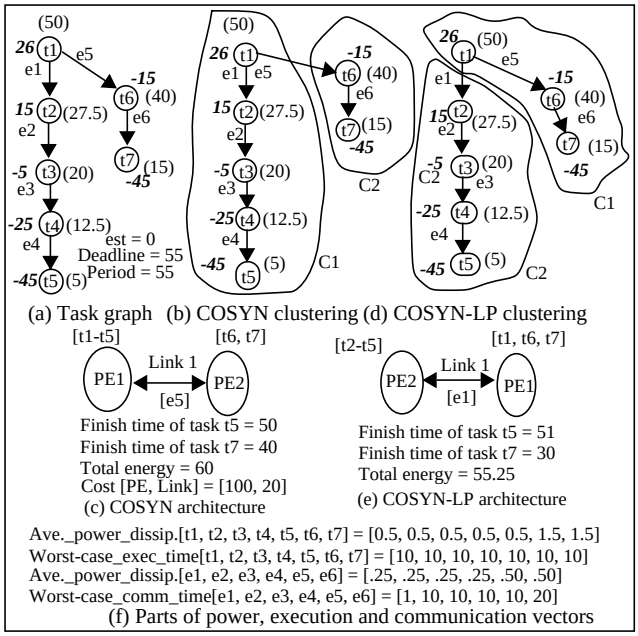

Figure 4. Clustering for low power

The energy-level based clustering technique generally does not result in a significant increase in the schedule length. This is due to the fact that energy and schedule length optimization are not necessarily divergent goals [12].

4.2 Cluster Allocation

We next discuss cluster allocation and finish-time/power estimation in the outer and inner loops of co-synthesis. In the outer loop of co-synthesis, the allocation array is created, as before, for each cluster. Each allocation is checked to see if the peak power dissipation as well as memory capacity (in case of general-purpose processor) of the associated PE/link is exceeded. Entries in the allocation array are ordered based on increasing average power dissipation. If there are more than one allocation with equal average power dissipation, then the one with the least dollar cost is chosen.

In the inner loop, during performance estimation, in addition to FTE, architecture power/energy estimation is also performed. The peak power dissipation, average energy dissipation, and average power dissipation for each processor, FPGA, ASIC, and communication link in the architecture are estimated as follows. **Processor/link:** The average and peak power are estimated based on the tasks (edges) allocated to the processor (link). The quiescent power dissipation of a processor (link) indicates the power dissipation during the idle time when no task (edge) is assigned to it. The power dissipation of a task (edge) is obviously higher than the quiescent power dissipation of a processor (link). Let $t_i \in \{T\}$ be the set of tasks assigned to the pth processor P. Similarly, let $e_j \in \{E\}$ be the set of communication edges assigned to the lth link L. The peak power for P is max{peak power ($t_i \in \{T\}$)}. Similarly, the peak power for L is max{peak power ($e_j \in \{E\}$)}. Let $\Re^{\xi}$ represent the average energy, and let θ^{ξ} represent the 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 are estimated as follows (the average power dissipation of P and L are estimated by dividing the total average energy dissipation by the hyperperiod).

Table 1: Experimental results for examples from the literature

Example/Number of tasks	Number of PEs/Cost ($)			Number of links			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	1/5	N/A	1/5	0	N/A	0	37.00	NA	0.20
Prakash & Parker (1)/9	1/5	1/5	1/5	0	0	0	3691.20	59.15	0.40
Prakash & Parker (2)/9	2/10	3/10	2/10	1	1	1	7.42 hrs	56.79	0.54
Prakash & Parker (3)/9	N/A	3/12	2/10	N/A	1	1	N/A	193.30	0.58
Prakash & Parker (4)/13	1/5	N/A	1/5	N/A	1	1	106.95 hrs	N/A	0.84
Yen & Wolf Ex/6	N/A	3/1765	3/1765	N/A	2	2	N/A	10.63	0.74
Hou & Wolf Ex1/20	N/A	2/170	2/170	N/A	1	1	N/A	14.96	5.10
Hou & Wolf Ex2/20	N/A	2/170	2/170	N/A	1	1	N/A	4.96	2.64
DSP/119	N/A	N/A	2/100	N/A	N/A	1	N/A	N/A	127.30

Table 2: Experimental results for transport systems

Example/ # of tasks	COSYN				COSYN-LP				Actual Transport System Power Dissipation
	No. of PEs/Cost($)	No. of links	CPU time (sec)	Average power dissipation(Watts)	No. of PEs/Cost($)	No. of links	CPU time (sec)	Average power dissipation (Watts)	Average Power dissipation (watts)
Transport System(1)/15	2/305	1	0.54	4.43	2/368	1	1.20	2.66	2.45
Transport System(2)/45	4/455	3	1.42	7.72	4/554	3	2.20	5.73	5.29
Transport System(3)/156	13/1725	11	118.40	26.40	13/1993	11	142.60	23.57	22.18

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

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

FPGA/ASIC: Tasks assigned to FPGAs and ASICs 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.

System power dissipation: The average power dissipation of the partial architecture is estimated by dividing the total estimated energy dissipated in PEs/links in it 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, similar to COSYN.

5 Experimental Results

Our co-synthesis algorithms, COSYN and COSYN-LP, are implemented in C++. Table 1 provides an overview of the examples on which we have run COSYN. Prakash & Parker(0-4) are from [4]. Prakash & Parker(0) is the same as task 1 in [4]. Prakash & Parker(1-3) are the same as task 2 in [4] with different constraints. Prakash & Parker(4) is a combination of task 1 and task 2 from [4]. Yen & Wolf Ex is from [6]. Hou & Wolf Ex(1,2) are from [7]. DSP is from [10] and its deadline and period were assumed to be 6500 ms. The PE and link libraries used in these results are the same as those used in the corresponding references. We also ran COSYN and COSYN-LP on various Bell Laboratories telecom transport system task graphs. These are large task graphs representing real-life field applications. The execution times and power dissipation for the tasks in the transport system task graphs were either experimentally measured or estimated based on existing designs. 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), two ASICs, one XILINX 3195A FPGA, and one ORCA 2T15 FPGA. The link library was assumed to contain a 680X0 bus, a 10 Mb/s LAN, and a 31 Mb/s serial link. As shown in Table 1, COSYN consistently outperforms both MILP [4] and iterative improvement techniques [6,7]. For Prakash & Parker(4), the MILP technique required approximately 107 hours 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 a second on Sparcstation 20 with 256MB RAM. Results in Table 2 show that COSYN was able to handle the large telecom transport system task graphs equally efficiently. COSYN-LP was able to reduce power dissipation by an average of 25% (this is the average of the individual cost reductions; the average is similarly determined for other parameters) over the basic COSYN algorithm at an average increase of 19% in cost. Also, as shown in the last column in Table 2, the actual system power measurements made on the COSYN-LP architectures indicate that the error of the COSYN-LP power estimator is within 9%.

6 Conclusions

We presented an efficient distributed system co-synthesis algorithm in this paper. Even though it is a heuristic algorithm, 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 the first co-synthesis algorithm for power optimization.

References

1. M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman & Co., 1979.
2. R. K. Gupta, *Hardware-Software Co-synthesis of Digital Systems, Ph.D. thesis*, Dept. of EE, Stanford University, 1994.
3. A. Kalavade and E. A. Lee, "A global criticality/local phase driven algorithm for constrained hardware/software partitioning problem," in *Proc. Int. Wkshp. Hardware-Software Co-Design*, pp. 42-48, Sept. 1994.
4. S. Prakash and A. Parker, "SOS: Synthesis of application-specific heterogeneous multiprocessor systems," *J. Par. & Dist. Comput.*, pp. 338-351, Dec. 1992.
5. 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, 1994.
6. T.-Y. Yen and W. Wolf, "Communication synthesis for distributed embedded systems," in *Proc. Int. Conf. Computer-Aided Design*, Nov. 1995.
7. J. Hou and W. Wolf, "Process partitioning for distributed embedded systems," in *Proc. Int. Wkshp. Hardware-Software Codesign*, pp. 70-76, 1996.
8. S. Srinivasan and N. K. Jha, "Hardware-software co-synthesis of fault-tolerant real-time distributed embedded systems," in *Proc. European Design Automation Conf.*, pp. 334-339, Sept. 1995.
9. E. Lawler and C. Martel, "Scheduling periodically occurring tasks on multiple processors," *Inform. Process. Letters*, vol. 12, Feb. 1981.
10. S. Yajnik, S. Srinivasan and N. K. Jha, "TBFT: A task based fault tolerance scheme for distributed systems," in *Proc. ISCA Int. Conf. Parallel & Distr. Comput. Syst.*, pp. 483-489, Oct. 1994.
11. B. P. Dave, G. Lakshminarayana, and N. K. Jha, "COSYN: Hardware-software co-synthesis of embedded systems," Tech. Rep., CE-J96-003, Dept. of EE, Princeton University, Oct.1996.
12. V. Tiwari, S. Malik and A. Wolfe, "Compilation techniques for low energy: An overview," in *Proc. Symp. Low-Power Electronics*, Oct. 1994.