CRUSADE: Hardware/Software Co-Synthesis of Dynamically Reconfigurable Heterogeneous Real-Time Distributed Embedded Systems

Bharat P. Dave
Bell Laboratories, Lucent Technologies, Holmdel, NJ 07733, USA

Abstract

Dynamically reconfigurable embedded systems offer potential for higher performance as well as adaptability to changing system requirements at low cost. Such systems employ run-time reconfigurable hardware components such as field programmable gate arrays (FPGAs) and complex programmable logic devices (CPLDs).

In this paper, we address the problem of hardware/ software co-synthesis of dynamically reconfigurable embedded systems. Our co-synthesis system, CRUSADE, takes as an input embedded system specifications in terms periodic acyclic task graphs with rate constraints and generates dynamically reconfigurable heterogeneous distributed hardware and software architecture meeting real-time constraints while minimizing the system hardware cost. We identify the group of tasks for dynamic reconfiguration of programmable devices and synthesize efficient programming interface for reconfiguring reprogrammable devices. Real-time systems require that the execution time for tasks mapped to reprogrammable devices are managed effectively such that real-time deadlines are not exceeded. To address this, we propose a technique to effectively manage delay in reconfigurable devices. Our approach guarantees that the real-time task deadlines are always met. To the best of our knowledge, this is the first co-synthesis algorithm which targets dynamically reconfigurable embedded systems. We also show how our co-synthesis algorithm can be easily extended to consider fault-detection and fault-tolerance.

Application of CRUSADE and its fault tolerance extension, CRUSADE-FT to several real-life large examples (up to 7400 tasks) from mobile communication network base station, video distribution router, a multi-media system, and synchronous optical network (SONET) and asynchronous transfer mode (ATM) based telecom systems shows that up to 56% system cost savings can be realized.

1 Introduction

Embedded systems perform application-specific functions using central processing units (CPUs) and application-specific integrated circuits (ASICs). ASICs can be based on standard cells, gate arrays or FPGAs or CPLDs. An embedded system architecture consists of hardware architecture and software architecture. Hardware architecture of an embedded system defines interconnection of various hardware components. Software architecture defines the allocation of sequence of codes to specific general-purpose processors. Hardware/ software co-synthesis is a process to obtain hardware and software architecture such that various embedded system constraints such as real-time, cost, power, *etc*., are met. Hardware/software co-synthesis involves various steps such as allocation, scheduling and performance estimation. Optimal hardware/software co-synthesis is known to be NP-complete problem [1]. Embedded systems employing reconfigurable hardware such as FPGAs and CPLDs are referred as reconfigurable embedded systems. Reconfigurable systems can provide higher performance as well as flexibility to adapt with changing system needs at low cost [2]-[4]. Dynamically reconfigurable embedded systems exploit reconfigurability of programmable devices at run-time to achieve further cost savings. With the availability of partially reconfigurable devices such as those from ATMEL AT6000 series and XILINX XC6200 series, dynamically reconfigurable systems have become viable [5]-[8]. However, dynamic reconfiguration of programmable devices adds additional complexity to already complex co-synthesis problem due to the identification and management of multiple reconfiguration programs for each programmable device in the system architecture.

Co-synthesis of heterogeneous distributed system have been previously addressed in [9]-[24]. Some of these co-synthesis systems [12,15,16,20,23,24] employ programmable devices such as FPGA. However, none of these systems target dynamically reconfigurable embedded systems.

Dynamically reconfigurable embedded system architectures requires dynamic reconfiguration of programmable hardware components such as FPGAs and CPLDs. These devices are either completely or partially reprogrammed at run-time to perform different functions at different times. Hardware/software co-synthesis of the dynamically reconfigurable architectures spans three major sub-problems: 1) Delay management, 2) Reconfiguration management, and 3) Reconfiguration controller interface synthesis. Delay for a circuit through a programmable device varies depending on how the constituent circuit is placed and routed. Delay management technique ensures that the delay constraint for the specific function is not exceeded while mapping the tasks to the programmable devices. Reconfiguration management technique identifies the grouping of tasks and their allocation such that the number of reconfigurations as well as time required for each reconfiguration is minimized while ensuring that the real-time constraints are met. Reconfiguration interface synthesis determines the efficient interface for reprogramming programmable devices such that cost of the system is reduced while minimizing the reconfiguration time.

We have developed a heuristic-based constructive co-synthesis algorithm, CRUSADE (Co-synthesis of ReconfigUrable System Architectures of Distributed Embedded systems) which optimizes the cost of the hardware architecture while meeting the real-time and other constraints. To the best of our knowledge, it is the first algorithm to address the co-synthesis of dynamically reconfigurable architectures. Fault-tolerant distributed embedded systems can offer high performance as well as dependability (reliability and availability) to meet the needs of critical real-time applications. We also show how CRUSADE can be easily extended to address needs of fault-tolerant systems. In order to establish its effectiveness, CRUSADE has been successfully applied to several large real-life examples from mobile communication network base station, video distribution router, and telecom embedded systems. However, in general, being heuristic, CRUSADE can never guarantee optimality.

The paper is organized as follows. Section 2 describes co-synthesis problem and preliminaries. Section 3 describes motivation behind dynamically reconfigurable architectures. Section 4 describes the challenges associated with co-synthesis of dynamically reconfigurable embedded system architectures and proposed techniques to address them. Section 5 provides overview of CRUSADE algorithm. Section

6 provides extension of CRUSADE to address fault-tolerant systems. Section 7 gives experimental results. Section 8 gives the conclusions.

2 Preliminaries

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

2.1 Problem Description

The embedded system functionality is generally described through a set of acyclic task graphs. This task based model is used in variety of prior co-synthesis systems [9,21,22,23,24]. Nodes of a task graph represent tasks and a directed edge between two communicating tasks indicate communication. Task are atomic units performed by embedded systems. A task contains both data and control flow information. Task graphs are required to be acyclic to reduce the complexity of the co-synthesis problem. However, there can be loops/cycles within a task. Each periodic task graph has an earliest start time (EST), period, deadlines as shown in Figure 1(a).

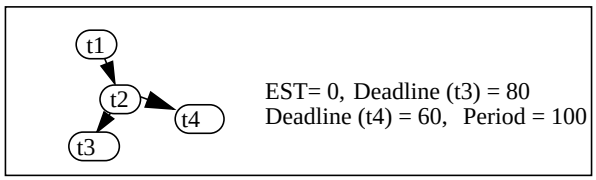

Figure 1. Task Graph

A co-synthesis problem can be summarized as follows. Given embedded system specifications in terms of acyclic task graphs, the objective is to find the hardware and software architecture such that the architecture cost is minimum while making sure that all real-time constraints are met.

2.2 Co-synthesis Framework

The architecture model: Our co-synthesis system does not employ a pre-determined (fixed) architectural template since such an approach can result in an expensive architecture and may not be suitable for a variety of embedded systems. In our co-synthesis system, the resulting embedded system can have a heterogeneous distributed architecture employing different types of processing elements (PEs) and links, where the architectural topology is not determined *a priori*. In the resulting architecture, there can be more than one configuration program for each FPGA/CPLD, *i.e*, it is time shared among multiple functions requiring reprogramming/ reconfiguration.

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

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

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

The execution model: Embedded system functions are specified by acyclic task graphs. Parameters used to characterize task graphs are described next. Each task is characterized by:

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

A *cluster* of tasks is a group of tasks which is always allocated to the same PE. Clustering of tasks in a task graph reduces the communication times and significantly speeds up the co-synthesis process [23]. Each cluster is characterized by the preference and exclusion vectors of its constituent tasks.

Each edge in the task graphs is characterized by:

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

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

Scheduling: We use a static scheduler which employs a combination of preemptive and non-preemptive scheduling to derive efficient schedules. Tasks and edges are scheduled based on deadline-based priority levels (see Section 5). The schedule for real-time task graphs is defined during architecture synthesis.

3 Motivation behind reconfigurable architectures

Reconfigurable architectures employ reconfigurable components such as FPGA and/or CPLD and are desirable for the following three major reasons.

1. In spite of heavy emphasis on simulation and regression testing, occasionally, design errors are indeed detected after the design is introduced in the market. If such design errors are found to be in FPGA/CPLD, then these devices can be reprogrammed in the field to prevent large expenses associated with recall of products as well as design upgrades in the factory.
2. Embedded system are generally released in the field with initial set of functions/features. At a later date, additional features or feature enhancements are offered to the customer. If the reconfigurable devices in initial release have sufficient resources and required connectivity to support additional features and/or feature enhancements, it would be possible to provide the required upgrade via simply reconfiguring the FPGAs and CPLDs.
3. Dynamic reconfiguration of FPGA/CPLD can result in low cost architectures due to temporal sharing of resources across multiple functions. To illustrate this, consider an example where there are 3 task graphs, *T1*, *T2* and *T3* required by an embedded system functions as shown in Figure 2(a). For simplicity, consider that resource library has two FPGAs, *F1* and *F2* as shown in Figure 2(b). *F1* can accommodate either *T1* and *T2* or *T1* and *T3* but not all three. On the other hand *F2* can accommodate all three task graphs if dynamic reconfiguration is employed. Execution times of task graphs on

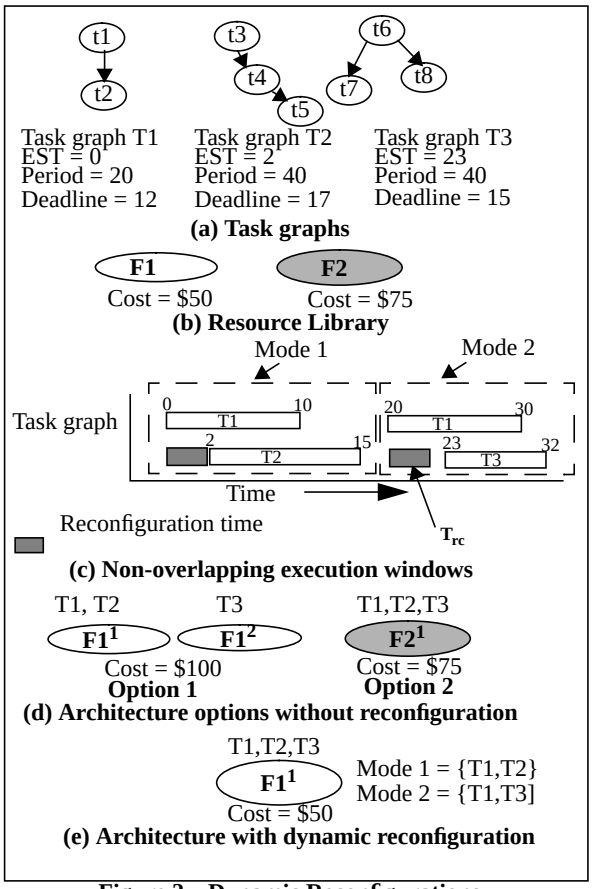

Figure 2. Dynamic Reconfigurations

resource library are shown in Figure 2(c). For simplicity, it is assumed here that execution times for each task graph are same on both FPGAs. However, in reality execution times will vary depending on the type of the FGPA. Hyperperiod, Γ, is computed as the least common multiple of periods of all task graphs. In traditional real-time computing, if P_i is period of task graph T_i, then $[\Gamma \div P_i]$ copies are obtained for it. In order to ensure feasibility of schedule, we need to ensure that real-time deadlines of all copies of all tasks in hyperperiod are mt. As shown in Figure 2(c), the execution times of task graphs 2 and 3 never overlap. Further, only two of the three functions are required at any time in the hyperperiod. Therefore, if reconfiguration of FPGAs is not employed, it would result in two architecture options shown in Figure 2(d). $F1^i$ is an ith instance of *F1*, and so on. If reconfiguration of FPGA is employed, architecture with one FPGA as shown in Figure 2(e) would have been sufficient. In such architecture, an FPGA will have two modes: *mode 1* and *mode 2*. In mode 1, the $F1^1$ will support task graphs *T1* and *T2* whereas in mode 2, it will support task graphs *T1* and *T3*.

4 Co-synthesis of Dynamically Reconfigurable Embedded Systems: Challenges and Solutions

In this section, we discuss challenges in co-synthesis of dynamically reconfigurable embedded systems and follow-up with techniques to address them.

4.1 Identification of non-overlapping task graphs

Challenge: Reconfigurable embedded systems are characterized by a set of task graphs whose execution slot do not overlap in time and therefore offer opportunities of realizing cost effective architectures by assigning multiple sets of task graphs to the same set of PPEs. The co-synthesis system needs to facilitate identification of such tasks graphs which can be assigned to the same PPEs.

Solution: In order to facilitate identification of non-overlapping task graphs, we define compatibility vector as follows.

Compatibility_vector of task graph $(T_i) = [\Delta_{i1}, \Delta_{i2}, ..., \Delta_{ik}]$ indicates compatibility of task graph T_i with other task graphs of embedded system. Δ_{ij} indicates compatibility of task graph T_i with task graph T_j. If $\Delta_{ij} = 0$, it implies that task graph T_i is compatible with task graph T_j and 1, if otherwise. If execution time of two task graphs do not overlap, they are said to be *compatible* task graphs and they can share the FPGA/CPLD resources. If two task graphs are not compatible, it implies that their respective execution time indeed overlap and therefore an independent set of FPGA/CPLD resources must be assigned. Generally, during task graph generation process, it is identified whether two task graphs are compatible with each other or not and that information relayed to the co-synthesis system by specifying the compatibility vector for each task graph. When compatibility vectors for task graphs are not specified, the co-synthesis system automatically identifies the non-overlapping task graphs based on start and stop times of tasks and edges following scheduling using the procedure shown in Figure 3.

Task graphs for which compatibility vector is not specified, we build architecture without requiring dynamic reconfiguration of the PPEs. Once the architecture is defined and deadlines are met, we identify merge potential of the architecture as summation of number of PPEs and links in the architecture. We create merge array that includes merge possibilities for each PPE. Each element of the merge array has tuple which specifies a pair of PPEs which can be merged into a single PPE with multiple modes resulting from dynamic reconfiguration. We pick each tuple and explore merge by creating multiple modes for PPE and follow-up with scheduling and finish time estimation. If deadlines are met, we accept the merge and use the modified architecture otherwise reject the merge and explore next merge from the merge array. Once all merges are explored, we compare modified architecture with the previous architecture, and if the architecture cost or merge potential is decreasing, we repeat the process. We stop the process when we can no longer reduce the architecture cost or merge potential.

4.2 Allocation of non-overlapping task graphs

Challenge: Once the non-overlapping task sets are identified, their allocation needs to be determined such that all real time constraints are met. During allocation, reconfiguration of a programmable device must be considered to exploit temporal sharing of a programmable device across multiple functions.

Solution: During allocation step, we create an allocation array which is an array of possible allocations at that point in co-synthesis. In allocation array, we provide multiple versions of each programmable device. Each version corresponds to different configuration of device which is also known as a mode of the device. A non-overlapping set of tasks is allocated to different version (mode) of the device. Once the architecture is defined, we merge the various versions of the device ensuring that real-time constraints are met.

Let us illustrate with an example. A cluster is a group of tasks which are allocated to same PE (clustering procedure is explained in Section 5). Clustering is performed to reduce allocation complexity and speed up co-synthesis algorithm at minimal cost impact on the architecture [23]. Consider four clusters *C[0]-[3]* as shown in Figure 4(a). Numbers next to cluster indicates its priority level (see Section 5). Further,

```
GENERATE_DYNAMIC_RECONFIGURATION(architecture,
 task graphs){
    current_arch = architecture;
    previous_arch_cost = previous_arch_merge_potential  = ∞;
    current_arch_merge_potential = number of PPEs;
    current_arch_cost = cost of current_arch;
    while(current_arch_cost  < previous_arch_cost OR
        current_arch_merge_potential <
        previous_arch_merge_potential){
        previous_arch_cost = current_arch_cost;
        previous_arch_merge_potential =
        current_arch_merge_potential;
        IDENTIFY_MERGE_TUPLES(current_arch,
        task graphs){
            for each PPE{calculate the merge potential
            with respect to rest of the PPEs;}
            for each PPE_i {PPE_i _tag = UNPAIRED;}
            for each unpaired PPE_j {
                merge_array = NULL;
                inter_PPE_tuple (PPE_j, PPE_k)  ← group PPE_j
                with one of the adjacent PPE, PPE_k,with
                which the merge potential is maximum;
                add inter_PPE_tuple (PPE_j, PPE_k) to merge_array;
                PPE_j_tag = PPE_k_tag = PAIRED;}
            for each element i of merge_array{i_tag = unexplored;}
            EXPLORE_MERGE{
                for each element j of merge_array{
                    inter_PPE_merge_array ← identify the
                    merge possibilities using architectural hints;
                    j_tag = explored;}
                for each element k of
                the inter_PPE_merge_array{k_tag = unexplored;}
                for each element l of inter_PPE_merge_array{
                    temp_arch ← current_arch is
                    modified considering l;
                    l_tag = explored;
                    perform merge ← create additional mode
                    for the FPGA and update download time;
                    run scheduler;
                    if (deadlines are met){
                        current_arch = temp_arch;
                        current_arch_merge_potential = number of
                        PPEs + number of links in current_arch;}
                }
            }
        }
    }
    return final_arch = current_arch;
}
```

Figure 3. The procedure for dynamic reconfiguration

assume that clusters *C1* and *C2* are non-overlapping, *i.e.*, execution window of tasks from *C1* do not overlap with tasks from *C2*. However, execution slots for tasks from *C3* do overlap with those of cluster *C1*. Assume that clusters *C1* through *C3* require an FPGA implementation. For allocation, higher priority clusters are allocated first. Therefore, *C0* is called for allocation first and the resulting partial architecture requiring CPU, RAM and ROM-A is shown in Figure 4(b). Next cluster *C1* is called for allocation. *C1* requires an addition of an $FPGA^1_1$ and ROM-B for storing associated programming interface as shown in Figure 4(c). $FPGA^i_j$ indicates *i*th instance and *j*th mode of the FPGA. Next cluster *C2* is called for allocation assuming deadlines are met. If the deadlines are not met, we explore next possible allocation as explained in Section 5.3. Since *C2* is non-overlapping with cluster *C1*, we create a new mode $FPGA^1_2$ as shown in Figure 4(d). Assuming that the deadlines are met, we call cluster *C3* for allocation. Cluster C3's execution slots do overlap with that of cluster *C1*, and therefore resources used for *C1* can not be time-shared with *C3*. Hence, *C3* is allocated to $FPGA^1_1$ to avoid a new mode as shown in Figure 4(e). Once all clusters are allocated, we explore merging of modes. In other words, we try to combine *C1*, *C2*, and *C3* in the same FPGA mode if there exists sufficient resources and deadlines are met. However, in this case since it is not feasible, architecture shown in Figure 4(e) is the final architecture.

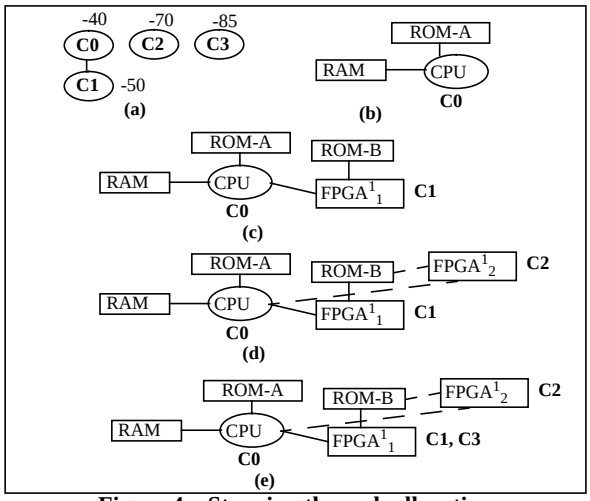

Figure 4. Stepping through allocation

4.3 Management of Multiple Software Images

Challenge: As explained before, various modes of programmable devices are created to facilitate time-sharing of resources. Each of this mode requires unique configuration software also known as software image. Switching between modes requires reconfiguration of a device. Time required to reconfigure a device is called boot time of the device. A co-synthesis system shall take into consideration the time required to reprogram the device while checking whether the deadlines will be met.

Solution: In order to address this, each programmable device is characterized by a *reboot_task*. This task is added at the beginning in each mode. Time required for *reboot_task* is determined by the type (serial or parallel) and speed of the programming interface. The boot time of the device is taken into consideration while estimating the finish time of the tasks to check whether deadlines are met. For example, task T_{rc} is added in front of *T3* as shown in Figure 2(c) to take into consideration time required for reconfiguration.

4.4 Reconfiguration Controller Interface Synthesis

Challenge: FPGAs/CPLD need to be programmed for correct operation. CPLDs are programmed via standard test port [5] used for boundary-scan testing. There are two different types of programming modes for FPGAs [6]-[8]: 1) serial, and 2) 8-bit parallel. Further each of these modes can be configured to be either Master or Slave. Master mode is used when FPGAs are programmed from a stand-alone PROM. Slave mode is used when the FPGA is programmed via CPU. Master mode can be used on power-up where as slave mode can be used in the field to provide upgrade of reconfiguration programs for bug fixes or providing new set of features. The speed of programming interface can vary from 1 MHz to 10 MHz (current technology). Also, when multiple programmable devices are used, they are generally chained to reduce the cost of the programming interface and share the PROM used for storing software image for various modes of various devices. The boot time of FPGAs/CPLDs can be as high as few hundred milliseconds which can be of concern for real-time systems requiring mode changes. Thus, there are several factors which

can determine the boot time for each device. Each of these options can impact boot time as well as cost and power of the system. Therefore, hardware/software co-synthesis system shall consider above aspects while synthesizing the correct programming interface for the programmable device.

Solution: For each embedded system, its boot time requirement is specified *a priori*. For each architecture option, reconfiguration option array is created. Each element of the reconfiguration option array indicates various options for programming. Each option in the configuration option array is characterized by boot time. Elements of the reconfiguration option array are ordered on the order of increasing dollar cost. We choose the one which has the lowest architecture cost while meeting the boot time requirements of the system. Boot time is recomputed based on the allocation and number of resources (CLBs/PFUs) that require reconfiguration.

4.5 Delay management

Challenge: Generally, all logic blocks of programmable devices, such as FPGAs and CPLDs, are not usable due to routing restrictions. A very high utilization of PFUs and pins may force the router to route the nets in such a way that it may violate the delay constraint, *i.e.,* the worst-case execution times defined by the execution time vector (defined in Section 2.2) may be exceeded. Therefore, hardware/software co-synthesis system shall manage delay through a programmable device such that delay constraint used during scheduling is not exceeded.

Solution: In order to address this aspect, we use two parameters: 1) effective resource utilization factor (ERUF), and 2) effective pin utilization factor (EPUF) to control the variations in delay once the functions mapped to the programmable device are synthesized and routed. We set ERUF equal to 70% and EPUF equal to 80%. These percentages were derived based on the existing designs and experimentally verified to guarantee the meeting of delay constraints during co-synthesis (see Section 7 for experimental results). Therefore, while allocating tasks to FPGAs/CPLDs, we ensure that we do not utilize more than 70% of resources (PFUs/CLBs/Flipflops) and 80% of the pins.

5 The CRUSADE Algorithm

In this section, we provide an overview of CRUSADE. Figure 5 presents one possible co-synthesis process flow which we follow in our work. This flow is divided up into three parts: pre-processing, synthesis, and dynamic reconfiguration generation. During pre-processing, we process the task graph, system constraints and resource library, and create necessary data structures. In traditional real-time computing theory, if $period_i$ is the period of task graph i then [hyperperiod÷ $period_i$] copies are obtained for it [9]. Co-synthesis algorithm must ensure that deadlines of all copies of all tasks in the hyperperiod are met. However, this is impractical from both co-synthesis CPU time and memory requirements points of view, specially for multi-rate task graphs where this ratio may be very large. We tackle this problem by using the concept of *association array* [23]. The *clustering* step involves grouping of tasks to reduce the search space for the allocation step [23]. Tasks in a cluster get mapped to the same PE. This significantly reduces the overall complexity of the co-synthesis algorithm since allocation is part of its inner loop. Our experience from [23] shows that task clustering results in up to three-fold reduction in co-synthesis CPU time for medium-sized task graphs with less than 1% increase in system cost. Our clustering technique addresses the fact there may be multiple longest paths through the task graph and the length of the longest path changes after partial clustering. We use the critical path task clustering method given in [23]. In order to cluster tasks, we first assign deadline-based priority levels to tasks and edges using the procedure from [23]. The priority level of a task is an indication of the longest path from the task to a task with a specified deadline in terms of computation and communication costs as well as the deadline. In the beginning, when allocation is not defined, we sum up the maximum execution and communication times along the longest path and subtract the deadline from the sum to determine the priority levels. However, priority levels are recomputed after each allocation as well as task clustering steps. In order to reduce the schedule length, we need to decrease the length of the longest path. This is done by forming a cluster of tasks along the current longest path. This makes the communication costs along the path zero. Then the process can be repeated for the longest path formed by the yet unclustered tasks, and so on.

The synthesis step determines the allocation of clusters. We define the priority level of a cluster as the maximum of the priority levels of the constituent tasks and incoming edges. Clusters are ordered based on decreasing priority levels. We pick the cluster with the highest priority level and create an allocation array.The synthesis part has two loops: 1) an *outer loop* for allocating each cluster, and 2) an *inner loop* for evaluating various allocations for each cluster. For each cluster, an *allocation array* consisting of the possible allocations at that step is created. While allocating a cluster to a hardware module such as an ASIC or FPGA, it is made sure that the module capacity related to pin count, gate count, *etc.*, is not exceeded. 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. After the allocation of each cluster, we recalculate the priority level of each task and cluster. Once the cluster is allocated, programming interface for the reconfigurable devices is synthesized.

The next step is *scheduling* which determines the relative ordering of tasks (edges) for execution (communication) and the start and finish times for each task (edge). We employ a combination of both preemptive and non-preemptive priority-level based static scheduling. Preemptive scheduling is used in restricted scenarios to minimize scheduling complexity. For task preemption, we take into consideration the operating system overheads such as interrupt overhead, context-switch, remote procedure call (RPC) *etc.* through a parameter called preemption overhead (this information is experimentally determined and provided *a priori*). Incorporating scheduling into the inner loop facilitates accurate performance evaluation. *Performance evaluation* of an allocation is extremely important in picking the best allocation. An important step of performance evaluation is *finish-time estimation*. In this step, with the help of the scheduler, the finish times of each task and edge are estimated using the longest path algorithm [23]. After finish-time estimation, it is verified whether the given deadlines in the task graphs are met. The *allocation evaluation* step compares the current allocation against previous ones based on total dollar cost of the architecture.

If deadlines are met, we explore merging of different modes of a programmable device as explained in Section 4.1 and 4.4 during *dynamic reconfiguration* generation phase.

6 Co-Synthesis of Fault Tolerant Systems

Embedded systems employed in critical application demand fault tolerance which provides fault detection followed by error-recovery. Such systems demand high dependability (reliability and availability) to meet the needs of critical real-time applications. For best results, hardware-software co-synthesis of such systems must incorporate fault tolerance during the synthesis process itself. We use the concepts from [24] to impart fault tolerance to the distributed embedded system architecture. Fault tolerance is incorporated by adding assertion tasks and duplicate-and-compare tasks to

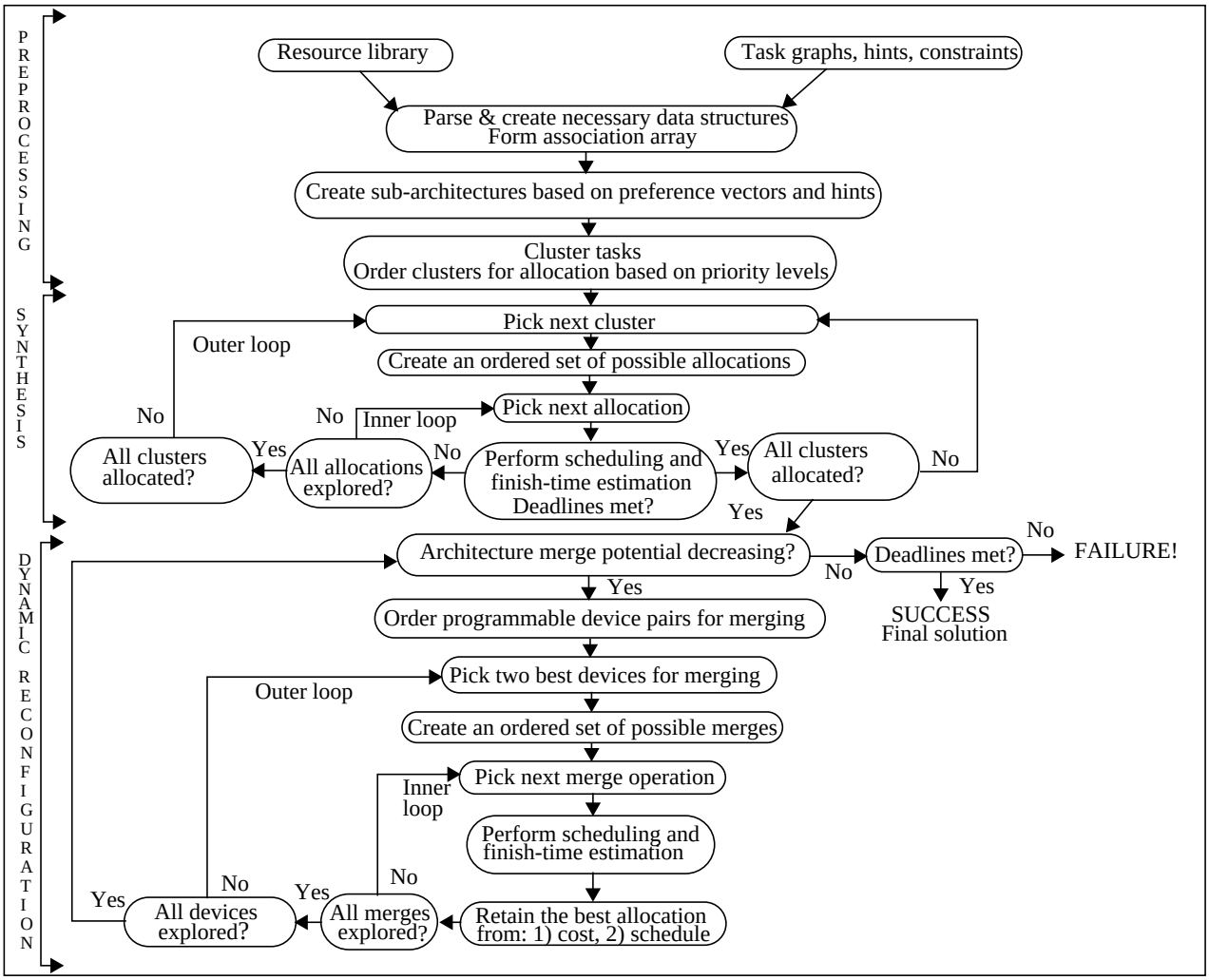

Figure 5. The co-synthesis process flow for CRUSADE

the system followed by error recovery. An assertion task checks some inherent property/characteristic of the output data from the original task. If that property is not satisfied, it flags the output data as erroneous. Some common examples of assertion tasks used in communication systems are: a) parity error detection, b) address range check, c) protection switch-control error detection, d) bipolar coding error detection, e) checksum error detection, *etc*. For each task, it is specified whether an assertion task is available or not. If not, the task is duplicated and the outputs of the two versions compared. For each assertion, an associated fault coverage is specified. It is possible that a single assertion is not sufficient to achieve the required fault coverage, and a combination of assertions is required. For each such task, a group of assertions and the location of each assertion is specified. For each check task (assertion or duplicate-and-compare task), the weight of the communication edge between the checked task and check task and the execution vector of the check task is specified. If a task is capable of transmitting any error at its inputs to its outputs, it is said to be error-transparent. This property is quite common. We exploit it to reduce the fault tolerance overhead. In order to facilitate dependability analysis, we require that the failure-in-time (FIT) rate, and mean-time-to-repair (MTTR) are specified *a priori* for each hardware and software module. The FIT rate indicates the expected number of failures in 10^9 hours of operation. Also, different functions of embedded systems can have different availability requirements. Therefore, we require that the availability requirements are specified for each task graph in the specification. Error recovery is enabled through a few spare PEs. In the event of failure of any service module (a set of PEs grouped together for replacement), a switch to a standby module is made for efficient error recovery. The basic co-synthesis process of CRUSADE is also used in its extension for fault tolerance, termed CRUSADE-FT. We describe next how various steps are modified.

Task clustering: We use the clustering technique [24] which exploits the error transparency property and determines the best placement of assertion and/or duplicate-and-compare tasks such that fault-detection times do not exceed associated constraints. This procedure is also used while creating the sub-architecture based on preference vectors. We assign the assertion overhead and fault tolerance level [24] to each task. We still use priority levels to identify the order of clustering for tasks. However, we use fault tolerance levels to cluster the tasks.

Inner loop of co-synthesis: For each allocation, in addition to the finish-time estimation, we explore whether any assertions need to be added, removed or shared following scheduling. We also obtain the service modules from the architecture using

architectural hints (if available, otherwise using an automated process [24]) and task graph availability requirements. Markov models are used to evaluate the availability of service modules and the distributed architecture.

Dynamic reconfiguration generation: While evaluating each possible PPE merge, we also perform: 1) addition/removal of assertions, as necessary, to reduce the fault tolerance overhead, and 2) dependability analysis to ensure that the resulting architecture continues to meet the availability constraints.

7 Experimental Results

Our co-synthesis algorithms CRUSADE and CRUSADE-FT are implemented in C++.

In order to test efficacy of our delay management technique through programmable PEs, we ran a series of experiments varying EPUF and ERUF from 70% to 100% while observing the impact on the delay constraint of various functional blocks. Sample results are shown in Table 1. It was observed that while setting EPUF = 80% and ERUF = 70%, delay constraints used for various functional blocks during co-synthesis process was not violated when those functions are actually synthesized with other functions on a device.

We also ran CRUSADE on communication system task graphs. These are large task graphs representing real-life field applications. These task graphs contain tasks from digital cellular communication network base station, video distribution router (video encoding/decoding using MPEG standard), synchronous optical network (SONET) interface processing, asynchronous transfer mode (ATM) cell processing, digital signal processing, provisioning, transmission interfaces, performance monitoring, protection switching, *etc*. These task graphs have wide variations in their periods ranging from 25 microseconds to 1 minute. The execution times were either experimentally measured or estimated based on existing designs. The general-purpose processors in the resource library had the real-time operating system, $pSOS^{+}$, running on them. The execution times included the operating system overhead. For results on these graphs, the PE library was assumed to contain Motorola microprocessors 68360, 68040, 68060, Power QUICC, (each processor with and without a 256KB second-level cache), 16 ASICs, XILINX 3195A, 4025, and 6700 series FPGAs, ATMEL AT6000 series FPGAs, XILINX XC9500 and XC7300 CPLDs, one ORCA 2T15 and 2T40 FPGAs. For each general-purpose processor, four DRAM banks providing up to 64 MB capacity were evaluated. DRAM devices with 60 ns access time were used. The link library was assumed to contain a 680X0 and Power QUICC buses, a 10 Mb/s LAN, and a 31 Mb/s serial link. Table 2 shows the experimental results. The first major column in this table gives characteristics of the distributed architecture derived by CRUSADE without employing dynamic reconfigurations of programmable devices, *i.e.*, each programmable device had only one mode. The CPU times are on Sparcstation 20 with 256 MB of DRAM. The system cost is the summation of the costs of the constituent PEs and links. The cost estimates of hardware components were based on yearly volume of 15K for the embedded system. The second major column gives the results with CRUSADE employing multiple modes for each of the programmable devices to time share the resources to reduce the cost. CRUSADE can realize up to 56% cost savings by employing dynamic reconfiguration of programmable devices.

For experiments with CRUSADE-FT, the FIT rates for various modules were either based on the existing designs or estimated using Bellcore guidelines [25]. MTTR was assumed to be two hours. The unavailability requirements for task graphs providing provisioning and transmission functions were assumed to be 12 minutes/year and 4 minutes/year respectively. The results with CRUSADE-FT are shown in Table 3. CRUSADE-FT can realize up to 53% cost savings by employing dynamic reconfiguration of programmable devices while deriving fault tolerant architectures.

8 Conclusions

We have presented an efficient co-synthesis algorithm for synthesizing dynamically reconfigurable heterogeneous distributed real-time embedded system architectures. Experimental results on various large real-life examples are very encouraging. To the best our knowledge, this is the first co-synthesis algorithm to tackle dynamically reconfigurable heterogeneous distributed embedded systems. We have also shown how fault tolerance considerations can be incorporated into our algorithm. For this case as well, the efficacy of algorithm was established through experimental results.

References

1. M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Co., 1979.
2. B. L. Hutchings and M. J. Wirthlin, “Implementation approaches for reconfigurable logic applications,” in Field Programmable Logic and Applications, Springer, Oxford, England, pp. 419-428, Aug. 1995.
3. J. E. Vuillemin, et. al., “Programmable active memories: Reconfigurable systems come of age,” IEEE Trans. VLSI Sys., vol. 4, pp. 56-69, Mar. 1996.
4. P. Athanas and K. L. Pocek, eds., *Proceedings The IEEE, Symposium on FPGAs for Custom Computing Machines*, IEEE Computer Society Press, Apr. 1995.
5. XILINX: *The ISP application guide and CPLD data book*, May 1997.
6. XILINX Inc., *XC6200 Field Programmable Gate Arrays*, Apr. 1997.
7. Atmel Corporation, *Configurable Logic Design and Application Book*, Aug. 1995.
8. Lucent Technologies, *FPGA data book*, October, 1996.
9. D. Kirovski and M. Potkonjak, "System-level synthesis of low-power hard real-time systems," in *Proc. Design Automation Conf.*, pp. 697-702, June 1997.
10. R. K. Gupta, *Co-Synthesis of Hardware and Software for Digital Embedded Systems*, Kluwer Academic Publishers, Norwell, Mass., 1995.
11. J. Henkel and R. Ernst, "A hardware/software partitioner using a dynamically determined granularity," in *Proc. Design Automation Conf.*, pp. 691-696, June 1997.
12. K. V. Rompaey *et al.*, "CoWare - A design environment for heterogeneous hardware/software systems," in *Proc. European Design Automation Conf.*, pp. 252-257, Sept. 1996.
13. F. Vahid and D. Gajski, "Incremental hardware estimation during hardware/software functional partitioning," *IEEE Trans. VLSI Systems*, vol. 3, no.3, pp. 459-464, Sept. 1995.
14. A. Kalavade and E. A. Lee, "The extended partitioning problem: Hardware/software partitioning and implementation-bin selection," in *Proc. Intl. Wkshp. Rapid Prototyping*, June 1995.
15. A. Kalavade and P. A. Subrahmanyam, "Hardware/ software partitioning for multi-function systems," in *Proc. Int. Conf. Computer-Aided Design*, pp. 516-521, Nov. 1997.
16. F. Balarin *et al*., *Hardware-Software co-synthesis of Embedded Systems: The POLIS Approach*, Kluwer Academic Publishers, Boston, MA, 1997.
17. M. B. Srivastava and R. W. Brodersen, "SIERA: A unified framework for rapid-prototyping of system-level hardware and software," *IEEE Trans. Computer-Aided Design*, pp. 676-693, June 1995.

18. D. D. Gajski, F. Vahid, S. Narayan, and J. Gong, "SpecSyn: An environment supporting the specify-explore-refine paradigm for hardware/software system design," Tech. Report CS-96-08, University of California, Riverside, CA, Sept. 1996.
19. J. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, "Ptolemy: A framework for simulating and prototyping heterogeneous systems," *Int. J. Comp. Simulation*, Jan. 1994.
20. K. Buchenrieder and C. Veith, "A prototyping environment for control-oriented HW/SW systems using state-charts, activity charts and FPGA's," in *Proc. European Design Automation Conf.*, pp. 60-65, Sept. 1994.
21. S. Prakash and A. Parker, "SOS: Synthesis of application-specific heterogeneous multiprocessor systems," *J. Parallel & Distributed Comput.*, vol. 16, pp. 338-351, Dec. 1992.
22. T.-Y. Yen and W. Wolf, "Communication synthesis for distributed embedded systems," in *Proc. Int. Conf. Computer-Aided Design*, pp. 288-294, Nov. 1995.
23. B. P. Dave, G. Lakshminarayana, and N. K. Jha, "COSYN: Hardware-software co-synthesis of distributed embedded systems," in *Proc. Design Automation Conf.*, pp. 703-708, June 1997.
24. B. P. Dave and N. K. Jha, "COFTA: Hardware-software co-synthesis of heterogeneous distributed embedded system architectures for low overhead fault tolerance," in *Proc. Int. Symp. Fault-Tolerant Computing*, pp. 339-348, June 1997.
25. Bellcore, "Generic reliability requirements for fiber optic transport systems," *Tech. Ref.* TR-NWT-00418, Dec. 1992.

Table 1: Experimental results for delay management through FPGAs/CPLDs

Circuit	No. of PFUs	Increase in delay (%), EPUF = 0.80						
		ERUF=0.70	ERUF=0.75	ERUF=0.80	ERUF=0.85	ERUF=0.90	ERUF=0.95	ERUF=1.00
cvs1	18	0.0	0.0	4.6	7.1	18.2	42.1	121.6
cvs2	20	0.0	2.5	6.1	8.3	22.6	68.7	138.9
xtrs1	36	0.0	8.9	9.3	9.8	28.1	46.2	88.6
xtrs2	40	0.0	10.4	12.6	18.6	24.8	53.6	72.1
rnvk	48	0.0	9.1	9.3	11.9	18.9	39.6	88.7
fcsdp	35	0.0	7.4	7.8	10.6	29.6	121.8	156.1
r2d2p	46	0.0	11.1	11.1	12.8	24.2	78.6	Not routable
cv46	74	0.0	9.2	10.4	11.9	22.8	62.1	Not routable
wamxp	84	0.0	12.1	14.6	18.1	28.6	54.7	Not routable
pewxfm	47	0.0	8.6	10.2	16.8	21.7	39.2	144.5

Table 2: Efficacy of CRUSADE

Example/ (No. of tasks)	CRUSADE without dynamic reconfiguration				CRUSADE with dynamic reconfiguration				
	No. of PEs	No. of links	CPU time (sec.)	Cost ($)	No. of PEs	No. of links	CPU time (sec.)	Cost ($)	Cost savings %
A1TR/(1126)	74	19	19322.6	26,245	61	16	20,473.4	16,225	38.2
VDRTX/(1634)	118	33	30,118.0	20,160	98	21	34,665.8	12,890	36.1
HROST/(2645)	244	48	68,771.6	34,898	219	36	77,125.4	24,100	30.9
EST189A/(3826)	334	87	82,664.7	48,445	312	68	91,705.3	33,815	30.2
HRXC/(4571)	388	93	89,183.4	51,170	348	74	104,045.6	37,900	25.9
ADMR/(5419)	406	102	112,629.1	64,885	375	93	124,118.1	40,005	38.3
B192G/(6815)	448	132	120,336.2	69,745	405	128	129,810.6	34,030	51.2
NG XM/(7416)	522	142	129,876.1	83,885	417	138	140,018.2	36,325	56.7

Table 3: Efficacy of CRUSADE-FT

Example/ (No. of tasks)	Fault tolerance without dynamic reconfiguration				Fault tolerance with dynamic reconfiguration				
	No. of PEs	No. of links	CPU time (sec.)	Cost ($)	No. of PEs	No. of links	CPU time (sec.)	Cost ($)	Cost savings %
A1TR/(1126)	98	28	22,800.6	30,815	74	21	24,487.8	21,355	30.7
VDRTX/(1634)	144	51	39,079.2	27,900	130	34	45,890.1	18,885	32.3
HROST/(2645)	361	88	85,690.6	52,830	275	59	97,550.4	33,075	37.4
EST189A/(3826)	470	116	105,943.1	64,965	398	85	123,540.2	43,115	33.6
HRXC/(4571)	512	131	110,968.9	60,688	446	108	131,627.7	41,930	30.9
ADMR/(5419)	526	136	134,559.8	79,025	474	136	158,864.7	50,810	35.7
B192G/(6815)	579	164	146,183.2	88,430	518	154	161,754.9	41,385	53.2
NG XM/(7416)	628	182	168,449.1	99,886	531	168	183,946.4	48,744	51.2