Hardware-Software Co-Synthesis of Low Power Real-Time Distributed Embedded Systems with Dynamically Reconfigurable FPGAs[‡]

Li Shang and Niraj K. Jha
Dept. of EE, Princeton University
{lshang, jha}@ee.princeton.edu

Abstract

In this paper, we present a multi-objective hardware-software co-synthesis system for multi-rate, real-time, low power distributed embedded systems consisting of dynamically reconfigurable FPGAs, processors, and other system resources. We use an evolutionary algorithm based framework for automatically determining the quantity and type of different system resources, and then assigning tasks to different processing elements (PEs) and task communications to communication links. For FPGAs, we propose a two-dimensional, multi-rate cyclic scheduling algorithm, which determines task priorities based on real-time constraints and reconfiguration overhead information, and then schedules tasks based on the resource utilization and reconfiguration condition in both space and time. The FPGA scheduler is integrated in a list-based system scheduler. To the best of our knowledge, this is the first multi-objective co-synthesis system, which uses dynamically reconfigurable devices to synthesize a distributed embedded system, to target simultaneous optimization of system price and power. Experimental results indicate that our method can reduce schedule length by an average of 41.0% and reconfiguration power by an average of 46.0% compared to the previous method. It also yields multiple system architectures which trade off system price and power under real-time constraints.

1. Introduction

Hardware-software co-synthesis entails automatic derivation of the hardware-software architecture of distributed embedded systems to satisfy multi-objective goals, such as performance, price and power. Allocation, assignment and scheduling are the three key steps in the hardware-software co-synthesis design flow. Allocation determines the type and number of PEs and communication links in the system architecture. Assignment determines the mapping of tasks (communications) to PEs (links). Scheduling determines the time when tasks and communications are executed.

An FPGA is a commonly used PE in distributed embedded systems. Compared with ASICs, FPGAs offer a parallel and flexible hardware platform. In order to reduce the reconfiguration overhead, many new reconfigurable architectures have been proposed [1]-[5]. In dynamically reconfigurable FPGAs, the embedded configuration storage circuitry can be updated selectively in a few clock cycles, without disturbing the execution of the remaining logic. Such FPGAs offer the potential for higher performance as well as the ability to efficiently support multi-mode requirements for embedded systems [6]. With the success of battery-based personal computing devices and wireless communication systems, low power has become a key issue in system design. Although its flexibility makes dynamically reconfigurable FPGAs a good solution for portable applications, the power consumption problem cannot be neglected. On-line reconfiguration not only introduces a delay overhead in task execution, but also a power overhead (which can account for half of the FPGA power consumption). This makes the FPGA power optimization problem more complex than that for general-purpose processors or ASICs.

1.1 Previous Work

The problem of dynamically reconfigurable FPGAs is addressed both in high-level synthesis [7]-[9] and system-level synthesis [16]-[21]. However, in system-level synthesis, the problem is much more complex. The execution time, power consumption, and reconfiguration overhead for each task and also the resource utilization and reconfiguration condition in the FPGA need to be considered. Allocation/assignment and scheduling, which are known be to NP-complete [10], need to be addressed in both the time and space domains.

Most hardware-software co-synthesis algorithms do not tackle FPGAs [11]-[15]. In those that do [16]-[21], system price is the single optimization objective. In [16], multiple tasks are not allowed to execute concurrently on the same FPGA. The approach in [18] uses mixed integer linear programming, which does not scale well to larger program sizes. Also, many algorithms make the simplifying assumption that the embedded system consists of just one processor and one FPGA [19]-[21].

1.2 Our Approach and Contributions

We use an evolutionary algorithm to tackle the problem of allocation and assignment. Such an algorithm has been shown to produce high-quality solutions in small run-times for the co-synthesis problem [14]-[16]. Multi-objective system requirements, such as price and power consumption, can be simultaneously optimized with this method. No limitation is imposed on the quantity of system resources. Since scheduling is performed in the inner loop of co-synthesis, a relatively accurate heuristic scheduler with a low time complexity is a must. Second,

[‡] Acknowledgements: This work was supported by DARPA under contract no. DAAB07-00-C-L516.

efficient methods for reducing the delay and power overheads of dynamic reconfiguration are required. We propose a two-dimensional multi-rate cyclic scheduling heuristic. Depending on the resource and reconfiguration information, the scheduler treats each task fairly and tries to globally minimize the reconfiguration overhead.

Our co-synthesis system simultaneously optimizes system price and power consumption under real-time constraints. Multiple non-dominated solutions are provided to the system designer with different trade-offs between system price and power.

The rest of this paper is organized as follows. In Section 2, we define the various terms and models used in our co-synthesis system. In Section 3, we present an overview of the co-synthesis tool. In Section 4, we describe the scheduling algorithm. We provide the experimental results in Section 5. Finally, we conclude in Section 6.

2. Preliminaries

In this section, we define the concepts and models used in our co-synthesis system.

2.1 Input Specification

The input specification is assumed to be in the form of a set of task graphs, as shown in Figure 1. A task graph is a directed acyclic graph, in which a node denotes a task while an edge between tasks represents data dependency and the amount of data transmitted. Each task graph has a period, which represents the interval between the earliest start times of its consecutive executions. In real-time systems, hard deadlines are associated with some of the tasks. An embedded system containing multiple task graphs with different periods is called multi-rate. The least common multiple (LCM) of the different task graph periods is defined as the *hyperperiod*. A valid static schedule is defined over a hyperperiod [22].

2.2 Resource Library Model

In addition to task graphs, a co-synthesis algorithm also needs to be fed information from a resource library. This library consists of general-purpose processors, dynamically reconfigurable FPGAs, communication links and memories that can be used for co-synthesis.

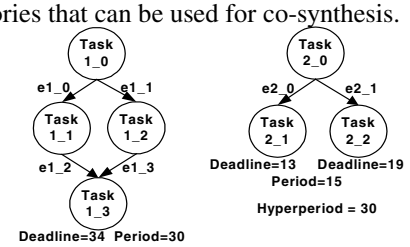

Figure 1: Task graphs

In dynamically reconfigurable FPGAs, a one-dimensional reconfiguration model is commonly used as shown in Figure 2 [1,3,5]. In this model, the atomic reconfiguration storage unit that can be dynamically updated is a frame. The reconfiguration of one frame does not disturb the execution of other frames. A task may reutilize a configuration pattern left behind by an earlier task. Multiple frames can only be reconfigured one by one. Each ready task needs to be loaded into contiguous frames in the FPGA reconfiguration memory before its execution. For each frame, the task has a specific configuration pattern. If the required configuration pattern cannot be found in the corresponding frame in the FPGA, a pattern miss is said to occur. Similar to caches in computers, compulsory, conflict, capacity and coherent misses can occur in the reconfiguration memory of FPGAs.

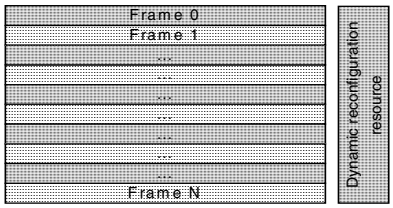

Figure 2: A dynamically reconfigurable FPGA mode

The following parameters are defined for each dynamically reconfigurable FPGA in the resource library: price, number of configuration frames, reconfiguration bandwidth, number of reconfiguration bits for each frame, number of I/Os, idle power, and reconfiguration power per frame. For each task, the worst-case execution time, average power consumption, and memory requirement to store reconfiguration and computation data on each FPGA type in the resource library are specified.

General-purpose processors are described by price and a variable indicating whether or not it has a communication buffer. For each task, the worst-case execution time, average power consumption, preemption time, and memory load are specified for each type of processor in the resource library. Communication links are described by price, packet size, average power consumption per packet, worst-case communication time per packet, pin requirement, idle power consumption, and contact counts. Memory blocks are modeled by price, power and size. The memory requirement for computation and communication is specified for each task.

The information for each task, such as execution time and power consumption etc., can be characterized with the help of techniques such as those presented in [23]-[26].

3. Hardware/Software Co-synthesis Overview

Allocation, assignment and scheduling are the three main steps that need to be carried out in co-synthesis. We use an evolutionary algorithm based framework for allocation and assignment [16]. However, in [16], only system price was minimized. Also, it used an FPGA model that supported the execution of only one task at a time. This model is not suitable for the current generation of FPGAs. Our co-synthesis system does not impose any restrictions on the quantity of different system resources. Thus, a combination of point-to-point links and buses connect the various PEs in a distributed system. We propose a new two-dimensional multi-rate scheduling algorithm for dynamically reconfigurable FPGAs in an embedded system. This aids the static system-level scheduler. Scheduling is discussed in detail in Section 4.

An overview of our co-synthesis system is shown in Figure 3. Co-synthesis solutions are organized in clusters. Solutions within a cluster share the same allocation, but

have different assignments. Solutions are initialized first. Then evolution operators, i.e., reproduction, mutation, and information trading, are used to transform allocation and assignment to obtain the next generation of solutions. Within each cluster, the assignment information may be mutated or traded between different solutions. Allocation information may be mutated or traded between different clusters. The rank of solutions is determined in a two-dimensional space: system price and power consumption. The Pareto-ranking method is used for this purpose. A solution's rank is equal to the number of other solutions that do not dominate it (a solution dominates another if it is better in both power consumption and system price). Finally, when a pre-specified number of generations has passed without improvement, invalid solutions, i.e., those that do not meet the deadlines, are pruned out, and the remaining non-dominated solutions are reported to the system designer.

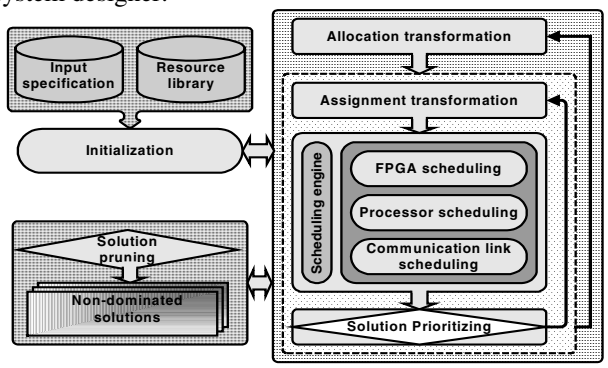

Figure 3: Hardware/software co-synthesis overview

4. Scheduling Algorithm

The static scheduling algorithm is invoked in the inner loop of co-synthesis after the allocation and assignment steps. Tasks (communication events) need to be scheduled on different processors and FPGAs (communication links). Processors and communication links represent a sequential resource. Hence, they require a one-dimensional scheduling problem to be solved. However, scheduling for dynamically reconfigurable FPGAs is a two-dimensional problem, including both the time and space domains, as described next.

1. Scheduling sequence: At each scheduling point, multiple ready tasks may reside in the candidate pool. Each task may have different time, resource and reconfiguration requirements, and power consumption. Thus, changing the scheduling order may have a significant impact on scheduling quality.

2. Location assignment policy: FPGAs are a parallel hardware platform. When a candidate task needs to be scheduled, there are many possible positions in the FPGA where the circuit implementing the task can be located. Assigning a task to a different location not only influences the current task, but may also impact the tasks scheduled either after or before it.

In this section, we dwell on the FPGA scheduling problem in significant detail.

4.1 Motivational Example

We next present an example to motivate our scheduling approach.

Example 1: Consider a system specification with three simple task graphs as shown in Figure 4. The allocation and assignment information for each task and communication event is shown in Table 1. Tasks 1_0 and 3_1 are assumed to have the same configuration patterns, while the configuration patterns for other tasks are assumed to be different. The reconfiguration time for each frame is 3.4 units. Based on the allocated PEs, the worst-case execution time for each task is shown in Table 2. The communication events C3_1 and C2_0 are executed on the bus that links the three PEs (in general, a more complex communication architecture can be synthesized). Their communication times are 15 and 10 units, respectively. Based on the traditional assumption in distributed computing, we assume that the communication time between two tasks assigned to the same PE is zero. Two different scheduling approaches are applied to these task graphs as described below (the first one based on prior work and the second one based on our work).

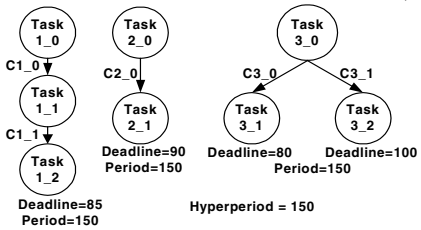

Figure 4: Task graphs

Table 1: Allocation and assignment information

Proc 1	Proc 2	FPGA	Bus
2_1	3_2	Other tasks	C2_0, C3_1

Table 2: Task execution time

Task	1_0	1_1	1_2	2_0	2_1	3_0	3_1	3_2
Worst-case exec. time	33	11	25	50	20	26	33	37

Scheduling approach I:

Scheduling sequence: The order of scheduling tasks is based on static slack-based priority [27]. The priority of task *i* is: $P_i = -(T\_latest\_ready_i - T\_earliest\_ready_i)$

where $T\_earliest\_ready_i$ is the earliest ready time of task *i* and $T\_latest\_ready_i$ is its latest ready time. These two values are computed by conducting a topological search of the task graphs based on as-soon-as-possible (ASAP) and as-late-as-possible (ALAP) scheduling.

Location assignment policy: Configuration patterns are allowed to be loaded into the FPGA before the task ready time. Configuration patterns left by earlier tasks can be utilized by later tasks. If there are several candidate positions in the FPGA where the task can be placed, the

heuristic is to find a position that allows the task to start as soon as possible. This location assignment policy is similar to the greedy heuristic proposed in [19].

Table 3 (first row) shows the schedule length, reconfiguration resource utilization (lower the better), and reconfiguration power consumption. The deadline is violated in this case. Figure 5 shows the FPGA, processor and bus schedule. The shaded blocks represent framewise reconfiguration. Reconfigurations introduced by compulsory misses are not shown, as they occur only once in the beginning of the first hyperperiod. The numbers in brackets indicate the sequence in which the tasks are scheduled.

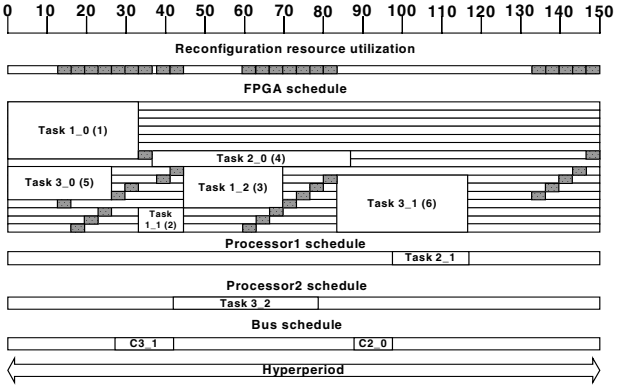

Figure 5: Scheduling result for Approach I

Scheduling approach II:

This is the approach we take.

Scheduling sequence: The order of scheduling tasks is determined dynamically by task priorities, which consider both real-time constraints and the reconfiguration overhead information (details given in Section 4.2).

Location assignment policy: The global reconfiguration information for all the tasks assigned to the FPGA is considered, as is the current state of the FPGA.

Table 3 (second row) and Figure 6 indicate the schedule quality for this approach. □

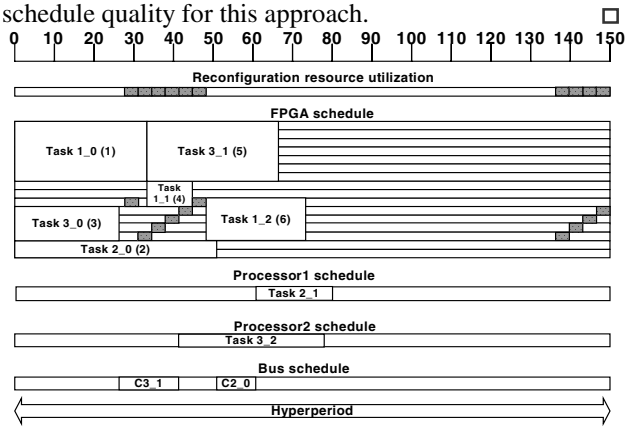

Figure 6: Scheduling result for Approach II

From the above example, we find, not surprisingly, that different FPGA scheduling policies may dramatically influence the scheduling quality, i.e., the satisfaction of real-time constraints, reconfiguration resource utilization, and reconfiguration power consumption. First, since reconfiguration itself consumes a significant amount of power, minimizing the reconfiguration overhead is important for reducing system power consumption. Second, solutions that cannot satisfy real-time constraints necessitate faster (and generally more expensive) PEs. This increases system price. A good scheduling approach reduces scheduling length and indirectly the system price and power consumption.

Table 3: Scheduling results

App.	Deadline	Schedule length	Reconfig. utilization	Reconfig. power
I	Violation	117	48%	127 mW
II	Satisfied	80	23%	61 mW

4.2 Two-Dimensional FPGA Scheduling Algorithm

In this section, we describe the two-dimensional scheduling algorithm for the dynamically reconfigurable FPGAs in the embedded system. Scheduling sequence and location assignment policy are the two important factors that need to be considered.

4.2.1 Scheduling Sequence

As in Approach I in Example 1, static slack-based priorities are commonly used to order tasks for scheduling on processors. The intuitive idea behind this approach is that a task with a longer slack can tolerate some delay and should yield to another task with a shorter slack. This approach works well on sequential resources. However, this approach is not suitable for FPGAs, which can execute multiple tasks concurrently. In the static slack-based priority approach, tasks along the critical path of one task graph may always be scheduled before tasks in other task graphs. This can prove to be quite sub-optimal for FPGAs. Our experimental results show that scheduling tasks from different task graphs in an interleaved fashion in FPGAs leads to better global schedules.

Another difference between processors and FPGAs is that in FPGAs, reconfiguration degrades performance and increases power consumption. Hence, in order to reduce the reconfiguration overhead, among the multiple ready tasks, those that can utilize the configuration patterns that already reside in FPGA should be preferred. This means that the reconfiguration overhead should also influence task priority. We propose a dynamic priority based approach, which dynamically updates the task priority, as follows.

$$priority_{task_i} = -latest\_finish\_time_{task_i} + exec\_time_{task_i} + reconf\_overhead_{task_i} - reconf\_inter_{task_{i,j}}$$

where $latest\_finish\_time_{task_i}$ is the latest possible finish time for task $task_i$ which is computed by conducting a backward topological search of the task graph based on the task graph deadline information. $exec\_time_{task_i}$ is the worst-case execution time for $task_i$ on the assigned PE. $reconf\_overhead_{task_i}$ is the reconfiguration overhead of $task_i$. $reconf\_inter_{task_{i,j}}$ is the inter-task reconfiguration time between adjacent tasks, which is updated dynamically, as follows. For each $task_i$ in the candidate pool that has the same configuration patterns as $task_j$, which has been removed from the candidate pool for scheduling on the

FPGA, the value of this variable is zero. In this approach, both the real-time constraints and reconfiguration overhead are considered, and tasks from different task graphs are treated fairly.

4.2.2 Location Assignment Policy

When a task is selected based on the above approach, multiple candidate locations may exist in the FPGA. The location assignment policy for a task not only influences the current task, but also the scheduling result for other tasks. Several factors need to be considered in the context, as discussed next.

Reconfiguration prefetch: Each task needs to be loaded into the FPGA first before starting its execution. When the task implementation is large, the reconfiguration overhead may be substantial even in dynamically reconfigurable FPGAs. Reconfiguration prefetch can be employed to alleviate this problem. The system can try loading the task earlier and finish the reconfiguration before the ready time of the task. This may allow the reconfiguration time for the task to be hidden.

Configuration pattern reutilization: When a new task needs to be loaded into an FPGA, its configuration patterns need to be mapped into a set of contiguous frames. If subsets of the requisite configuration patterns already reside in the FPGA, loading of those data can be avoided. This helps reduce the reconfiguration overhead.

Eviction candidate: If not enough free space is left in the FPGA for new configuration patterns, some existing configuration patterns need to be evicted from the device. This problem is similar to the paging problem [28] and the weighted caching problem [29]. However, for our problem, all the frames assigned to a task need to be contiguous, which makes the problem more complex. The frames that need to be reconfigured for the incoming task may contain configuration patterns from different tasks, each executing at a different recurrent frequency (this is the number of times the task executes in the hyperperiod). When a configuration pattern with a higher recurrent frequency is evicted, it may introduce a new reconfiguration overhead later in the hyperperiod. We define the eviction cost for a candidate position for this task based on a weighted sum of all the configuration patterns that need to be replaced, as follows:

$$eviction\_cost = \sum_{i=start\_frame}^{end\_frame} recurrent\_freq_{frame_i}$$

where $recurrent\_freq_{frame_i}$ is the recurrent frequency of the configuration pattern in $frame_i$. The *eviction_cost* is the weighted cost for this candidate position. The candidate positions with lower *eviction_cost* should be preferred.

Fitting policy: The algorithm should try to avoid fragmentation of the FPGA configuration memory when choosing the candidate position from the FPGA.

Slack time utilization: Some of the possible candidate positions for a ready task may already have configuration patterns similar to the newly required ones. Using these positions would lower the eviction cost. However, the task may not be able to start execution immediately if assigned to such candidate positions. A greedy policy may neglect such candidate positions. This may adversely impact the schedule quality for other tasks. This is because reconfiguration hardware is a sequential resource. Reconfiguration of one frame delays reconfiguration of others. Therefore, reconfiguration overhead minimization should have a high priority. Thus, a better approach to the candidate position selection problem is to possibly choose a slightly inferior solution for the given task which helps find a better global solution.

The slack of a task indicates to what extent an inferior solution can be tolerated for it. Since the task may share the slack with other tasks, which may not have been scheduled yet, the slack should not be completely used up by the current task. The portion of the slack, which can be utilized for the task in question, should be the slack divided by the depth of the sub task graph (the root vertex of the sub task graph is the current task,), as follows:

$$tolerate\_start\_time_j = start\_time_j + \frac{slack_{task_j}}{depth_{sub\_graph}},$$

where $start\_time_j$ is the ready time of $task_j$, $depth_{sub\_graph}$ is the depth of the sub task graph in terms of the number of tasks, and $slack_{task_j}$ is the slack of $task_j$. $tolerate\_start\_time_j$ is the delayed start time that $task_j$ can tolerate.

Our FPGA location selection policy is based on the above analyses. The influence of reconfiguration overhead on the dispatch time of each task is minimized. Candidate positions with lower weighted reconfiguration overhead and tolerable delay are always chosen. Reconfiguration data can be effectively shared among tasks with similar reconfiguration patterns. The reconfiguration overhead is, therefore, effectively reduced and sometimes hidden. This also minimizes reconfiguration power, a significant part of the power consumption in FPGAs.

4.2.3 The Algorithm

The pseudo-code for the two-dimensional scheduling algorithm is shown in Figure 7. First, root nodes from all the task graphs are put into the candidate pool (line 2). The priority of each task in the candidate pool is updated dynamically (line 4), and the task, $task_i$, with the highest priority chosen (line 5). Since the parent tasks of $task_i$ may be assigned to PEs other than $task_i$ itself, the corresponding communication events need to be scheduled on the communication resource first (line 6). Then $task_i$ is scheduled on the candidate PE (line 7). Finally, scheduling of $task_i$ leads to other tasks becoming ready (line 8). The key part of the scheduling algorithm is *schedule_task*($task_i$), whose working is illustrated next.

```
1. scheduling_algorithm(){
2.   candidate_pool ← root_nodes
3.   while(pending_tasks ≠ NULL){
4.       priority_calculation(candidate_pool)
5.       task_i ← extract(candidate_pool)
6.       sched_input_communication(task_i)
7.       schedule_task(task_i)
8.       candidate_pool ← introduce_ready_task(task_i)}}
```

Figure 7: Pseudo-code of the scheduling algorithm

Consider task *C* in the partial FPGA schedule shown in Figure 8. When this task is being loaded into the FPGA, the reconfiguration overhead may be introduced before or after the task, shown as shaded blocks.

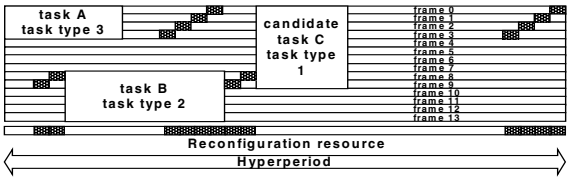

Figure 8: A task scheduling example

Two issues need to be considered for the reconfiguration blocks introduced before task *C*. First, the timespans of the empty slots in the different frames among the possible candidate positions for task *C* may be different. Since the reconfiguration hardware is a sequential resource, reconfiguration of one frame will delay the reconfiguration of other frames and even the start time of the task. Second, the reconfiguration slots left unused between the reconfiguration events and task *C* cannot be utilized by tasks with different configuration patterns. In our approach, the priority, P_{frame_i}, to determine the reconfiguration sequence of frames is defined as follows:

$$P_{frame_i} = \begin{cases} -(r\_t_{task} - s\_t_{frame_i}), & r\_t_{task} \geq s\_t_{frame_i} \\ -(hyperperiod - s\_t_{frame_i} + r\_t_{task}), & r\_t_{task} < s\_t_{frame_i} \\ r\_t_{task} = ready\_time_{task} \text{ modulo } hyperperiod \\ s\_t_{frame_i} = start\_time_{frame_i} \text{ modulo } hyperperiod \end{cases}$$

where $ready\_time_{task}$ is the ready time of the task, $start\_time_{frame_i}$ is the start time of the empty time slot in $frame_i$. The idea is that if the duration between the reconfiguration slot start time and the task ready time is short, reconfiguration of the corresponding frame needs to be scheduled first. Otherwise, reconfiguration may not be completed by the task ready time and hence delay task execution. The reconfiguration slots in each frame are scheduled before this ready task based on a nonincreasing priority order. In order to hide the reconfiguration overhead whenever possible, a function called *schedule_back*() is used. This function looks backward for the first available reconfiguration slot from r_t_{task} to $s\_t_{frame_i}$ in the current frame. If the function returns false, it means that reconfiguration cannot start during $[s\_t_{frame_i}, r\_t_{task}]$. In this case, another function *schedule_front*() is invoked. This looks for the first available reconfiguration slot in the current frame from r_t_{task} to the finish time of the empty timespan. With this approach, the reconfiguration events are scheduled as soon as possible before the task ready time and also as closely as possible to this task, addressing both the issues raised before. In Figure 9, before candidate task *C*, frames 8 and 9 are scheduled first then frames 0 to 3.

We next discuss the issues involved in scheduling reconfiguration slots after the task. To leave enough flexibility for future tasks, the reconfiguration slots need to be placed as close to the next task as possible. Also, a priority needs to be defined to determine the scheduling order for all the needed frames in order to tackle the interrelationships among them, as follows:

$$P_{frame_i} = \begin{cases} -(f\_t_{frame_i} - r\_t_{task}), & f\_t_{frame_i} \geq r\_t_{task} \\ -(hyperperiod - r\_t_{task} + f\_t_{frame_i}), & f\_t_{frame_i} < r\_t_{task} \\ r\_t_{task} = ready\_time_{task} \text{ modulo } hyperperiod \\ f\_t_{frame_i} = finish\_time_{frame_i} \text{ modulo } hyperperiod \end{cases}$$

where $finish\_time_{frame_i}$ is the finish time of the empty timespan in $frame_i$. Function *schedule_back*() is called for each frame based on a nonincreasing priority order. It chooses the first available reconfiguration slot from $f\_t_{frame_i}$ to r_t_{task} in the current frame. With this approach, in Figure 8, in frames 0 to 3, the reconfiguration slots after task *C* are scheduled close to task *A* (note that tasks repeat after the hyperperiod). In frames 8 and 9, the reconfiguration slots are scheduled close to task *B*.

Function *schedule_task*($task_i$) contains two steps. First, *candidate_position_sort*($task_i$) calculates the priority for each candidate position. Its pseudo-code is shown in Figure 9. In lines 3 to 6, the algorithm calculates the priority of the frames in each candidate position. Then, for each candidate position, it schedules reconfiguration slots before the task based on the frame priorities (lines 7-9). From all the frames in this candidate position, it chooses the latest reconfiguration finish time to be the actual task ready time for this position. Then it uses the location assignment policy described in Section 4.2.2 to calculate the priority for each candidate position (line 10). Second, function *schedule_task_p*($task_i$) is invoked to schedule the task. Its pseudo-code is shown in Figure 10. The candidate position with the highest priority is chosen from *candidate_position_pool* (line 3). The reconfiguration slots before the task are scheduled first (line 4), then the reconfiguration slots after the task (line 5). Finally, the task itself is inserted into the schedule (line 6). If any of these three steps fails, the frame at which the failure occurs is chosen. The next time slot is searched from this frame, and using this frame a new priority for the candidate position is calculated (line 9). The candidate position is inserted into the priority queue at the appropriate location (line 10). Then a new candidate position is chosen to try to schedule the task (line 11).

For the FPGA scheduling algorithm, the time complexity is $O(n^2 \log n)$, where n is the number of tasks. However, in the average case, it behaves like an $n\log n$ algorithm.

```
1.  candidate_position_sort(task_i){
2.    for(i = 0; i < num_candidate_positions){
3.      for(j = position_start_i; j ≤ position_finish_i){
4.        slot_j = candidate_time_slot_find()
5.        slot_priority_j = slot_priority_calculation(slot_j)
6.        slot_priority_pl_i.insert(slot_priority_j)}
7.      for(j = slot_priority_pl_i.begin; j < slot_priority_pl_i.end){
8.        if(reconfig_frame_j = false){
9.          schedule_reconfig()}
10.     update_position_priority(candidate_position_i)}}}
```

Figure 9. Candidate position priority calculation

4.3 Scheduling Algorithms for Other Resources

FPGA scheduling is compatible with scheduling for processors and communication links. We use the same approach to schedule tasks (communication events) on different processors (communication links). The only difference is reconfiguration times can be made zero for processors and links, and the scheduling problem is one-dimensional (analogous to having only one frame in the FPGA).

```
1. schedule _task _p(task_i){
2.   while(candidate _position _pool ≠ NULL){
3.     candidate _position ← extract(candidate _position _pool)
4.     schedule _reconfig _before _task(task_i)
5.     schedule _reconfig _after _task(task_i)
6.     schedule _task _exec(task_i)
8.     if(false){
9.       calculate _priority(candidate _position.next _slot())
10.      candidate _position _pool.insert(candidate _position)
11.      next _candidate _position _chosen()
12.      continue }}}
```

Figure 10. Task scheduling

5. Experimental Results

In this section, we present experimental results for our FPGA scheduling algorithm and the hardware/software co-synthesis system. The system is implemented in C++ using the standard template library (STL). The resource library consists of various system resources available from the industry and academia. We use processors, memory blocks and communication links provided in [30]. The parameters of our dynamically reconfigurable FPGA model are based on Xilinx Virtex-E FPGAs [5]. The task graphs, which are input to the co-synthesis system, are generated by TGFF [30]. All the experiments were performed on a Pentium-III 667MHz PC (512MB memory) running Linux OS.

We first demonstrate the performance of our FPGA scheduling algorithm. We compare the results of scheduling for Approach I (Sect. 4.1), which is based on static slack-based priority, configuration prefetch, and pre-configuration utilization [19], and our Approach II. The results are shown in Table 4, which includes schedule length, reconfiguration power consumption and CPU time. Compared with Approach I, the improvements in schedule length and reconfiguration power are shown in columns 4 and 7, respectively, and also in Figure 11. For these examples, the number of task graphs varies from 4 to 6, and the total number of tasks in these task graphs is around 200. In Figure 11, the bars represent schedule length and the lines represent reconfiguration power.

Table 4: FPGA scheduling results

Ex.	Schedule length (in 10^3 time units)			Reconf. power (mW)			CPU time (seconds)	
	I	II	Imp.	I	II	Imp.	I	II
1	4815	1625	66.3%	101.4	12.0	88.2%	3.2	2.2
2	12530	5302	57.7%	186.7	88.1	52.8%	0.7	0.3
3	8353	5488	34.3%	114.8	81.3	29.2%	7.5	3.6
4	5992	2392	60.1%	88.4	37.3	57.8%	3.2	1.4
5	9139	6903	24.5%	120.2	94.0	21.8%	5.9	4.3
6	3282	2852	13.1%	223.3	193.3	13.4%	1.2	1.1
7	2066	1351	34.6%	33.1	19.9	39.9%	2.4	1.5
8	4270	1600	62.5%	99.3	33.1	66.7%	0.7	0.5
9	4600	4717	-2.5%	67.9	74.7	-10.0%	3.8	3.2
10	6444	2588	59.8%	110.3	0	100%	0.5	0.3

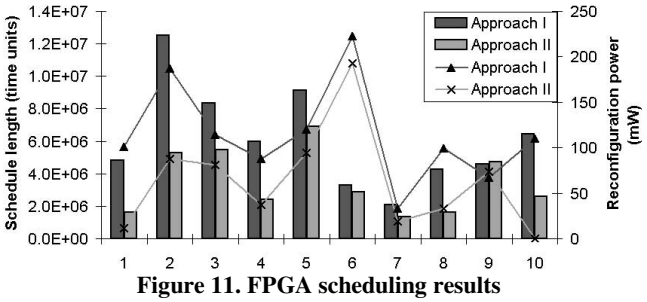

Figure 11. FPGA scheduling results

As opposed to Approach I, our algorithm always meets the real-time constraints (for Approach I, only solutions for Examples 3, 5 and 9 meet the real-time constraints). The average reduction in schedule length is 41.0% and the average reduction in reconfiguration power is 46.0%. Recall that reconfiguration power is frequently of the same order as the task power consumption. Hence, it is very important to reduce reconfiguration power. Reduction of the schedule length helps the co-synthesis system choose lower cost (and potentially slower) PEs without violating the real-time constraints, thus reducing the system price. In Example 9, our approach gets worse results. The reason is that in this example, because of the tight FPGA resource constraints, not much flexibility is left for our scheduling algorithm to explore the globally optimal solution. Since our approach may not choose a locally optimal solution for each task, it may at times get a worse result than Approach I which is much more greedy. Also, our algorithm needs slightly less run-time. This is because our algorithm looks ahead to the needs of future tasks and makes it easier to schedule them. Since Approach I is greedy and makes locally optimal choices, it needs more time to schedule tasks encountered later.

The results for our hardware/software co-synthesis system are shown in Table 5. In this table, columns 2 and 3, respectively, show the corresponding system price and power consumption of all the non-dominated solutions, and the last column shows the CPU time for co-synthesis. The system price is calculated by summing up the price of all the processors, FPGAs, communication links and memory in the distributed embedded system that is synthesized. The system power consumption is calculated by summing up all the execution, reconfiguration, communication and idle energies in the hyperperiod and dividing by the hyperperiod. Table 5 illustrates the ability of our co-synthesis system to effectively explore the design space. Our multi-objective optimization approach achieves a good trade-off between system price and power consumption. All real-time constraints are satisfied. The run-time indicates that large task graphs can be handled in a reasonable amount of time.

6. Conclusions

We presented a multi-objective hardware/software co-synthesis system for real-time distributed embedded systems. A novel two-dimensional multi-rate cyclic scheduling algorithm was proposed to tackle the scheduling problem in dynamically reconfigurable FPGAs. This algorithm not only minimizes schedule length (thus allowing cheaper PEs), but also significantly

reduces reconfiguration power. Reconfiguration power is the main bottleneck in exploiting the reconfiguration capability of modern dynamically reconfigurable FPGAs. Ours is the first co-synthesis system to target both price and power optimization in distributed embedded systems containing dynamically reconfigurable FPGAs.

Table 5: Hardware/software co-synthesis results

Example	Price (dollar)	Power consumption (mW)	CPU time (minutes)
1	209 389	144.7 66.1	99.7
2	42 212	394.5 253.6	133.6
3	57 153 173 198 525	619.7 305.5 271.1 121.9 108.4	19.8
4	159 174 209	745.5 626.9 503.6	54.2
5	153 385 420	815.6 699.8 489.4	28.8
6	232 367 394	922.7 829.6 557.5	14.9
7	156 353	684.2 462.9	3.0
8	156 204	790.5 345.6	18.0
9	209 238 250	852.0 345.8 265.3	39.2
10	156	353.8	2.1

References

[1] J. Hauser and J. Wawrzynek, "Garp: A MIPS processor with a reconfigurable coprocessor," in *Proc. Symp. Field-Programmable Custom Computing Machines*, pp. 12-21, Apr. 1997.

[2] S. Trimberger, D. Carberry, A. Johnson, and J. Wong, "A time-multiplexed FPGA," in *Proc. Symp. Field-Programmable Custom Computing Machines*, pp. 22-28, Apr. 1997.

[3] Z. A. Ye, A. Moshovos, S. Hauck, and P. Banerjee, "CHIMAERA: A high-performance architecture with a tightly-coupled reconfigurable functional unit," in *Proc. Int. Symp. Computer Architecture*, pp. 225-232, June 2000.

[4] T. Fujii et al., "A dynamically reconfigurable logic engine with a multi-context/multi-mode unified-cell architecture," in *Proc. Int. Solid-State Circuits Conf.*, Feb. 1999.

[5] Virtex-E data sheet, http://www.xilinx.com.

[6] Y. Shin, D. Kim, and K. Choi, "Schedulability-driven performance analysis of multiple mode embedded real-time systems," in *Proc. Design Automation Conf.*, pp. 495-500, June 2000.

[7] Z. Li, K. Compton, and S. Hauck, "Configuration caching techniques for FPGA," in *Proc. Symp. Field-Programmable Custom Computing Machines*, Apr. 2000.

[8] X. Tang, M. Aalsma, and R. Jou, "A compiler directed approach to hiding configuration latency in chameleon processors," in *Proc. Int. Conf. Field Programmable Logic and Applications*, pp. 29-38, Aug. 2000.

[9] M. Kaul, R. Vemuri, S. Govindarajan, and I. Ouaiss, "An automated temporal partitioning and loop fission approach for FPGA based reconfigurable synthesis of DSP applications," in *Proc. Design Automation Conf.*, pp. 616-622, June 1999.

[10] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company, NY, 1979.

[11] S. Prakash and A. Parker, "SOS: Synthesis of application-specific heterogeneous multiprocessor systems," *J. Parallel & Distributed Comput.*, vol. 16, pp. 338-351, Dec. 1992.

[12] T.-Y. Yen and W. Wolf, "Communication synthesis for distributed embedded systems," in *Proc. Int. Conf. Computer-Aided Design*, pp. 703-708, June 1997.

[13] 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.

[14] R. P. Dick and N. K. Jha "MOGAC: A multiobjective genetic algorithm for hardware-software co-synthesis of distributed embedded systems," *IEEE Trans. Computer-Aided Design*, vol. 17, pp. 920-935, Oct. 1998.

[15] E. Zitzler and L. Thiele, "Multiobjective evolutionary algorithms: A comparative case study and the strength pareto approach," *IEEE Trans. Evolutionary Computation*, vol. 3, no. 4, pp. 257-271, Nov. 1999.

[16] R. P. Dick and N. K. Jha, "CORDS: Hardware-software co-synthesis of reconfigurable real-time distributed embedded systems," in *Proc. Int. Conf. Computer-Aided Design*, pp. 62-68, Nov. 1998.

[17] B. P. Dave, "CRUSADE: Hardware/software co-synthesis of dynamically reconfigurable heterogeneous real-time distributed embedded systems," in *Proc. Design, Automation & Test in Europe Conf.*, pp. 97-104, Mar. 1999.

[18] N. Shenoy, A. Choudhary, and P. Banerjee, "An algorithm for synthesis of large time-constrained heterogeneous adaptive systems," *ACM Trans. Design Automation of Electronic Systems*, vol. 6, no. 2, pp. 207-225, Apr. 2001.

[19] B. Jeong, S. Yoo, S. Lee, and K. Y. Choi, "Hardware-software cosynthesis for run-time incrementally reconfigurable FPGAs," in *Proc. Asia & South Pacific Design Automation Conf.*, pp.169-174, Jan. 2000.

[20] Y. B. Li, T. Callahan, E. Darnell, R. Harr, U. Kurkure, and J. Stockwood, "Hardware-software co-design of embedded reconfigurable architectures," in *Proc. Design Automation Conf.*, pp. 507-512, June 2000.

[21] J. Noguera and R. Badia, "A HW/SW partitioning algorithm for dynamically reconfigurable architectures," in *Proc. Design, Automation & Test in Europe Conf.*, pp. 729-734, Mar. 2001.

[22] E. L. Lawler and C. U. Martel, "Scheduling periodically occurring tasks on multiple processors," *Information Processing Letters*, vol. 7, pp. 9-12, Feb. 1981.

[23] S. Malik, M. Martonosi, and Y.-T. Li, "Static timing analysis of embedded software," in *Proc. Design Automation Conf.*, pp. 147-152, June 1997.

[24] K. S. Khouri, G. Lakshminarayana, and N. K. Jha, "High-level synthesis of low power control-flow intensive circuits," *IEEE Trans. Computer-Aided Design*, vol. 18, no. 12, pp. 1715-1729, Dec. 1999.

[25] V. Tiwari, S. Malik, and A. Wolfe, "Power analysis of embedded software: A first step toward software power minimization," *IEEE Trans. VLSI Systems*, vol. 2, no. 4, pp. 437-445, Apr. 1994.

[26] S. Gupta and F. Najm, "Power modeling for high-level power estimation," *IEEE Trans. VLSI systems*, vol. 8, no. 1, pp. 18-29, Feb. 2000.

[27] B. Dave, G. Lakshminarayana, and N. K. Jha, "COSYN: Hardware-software co-synthesis of embedded systems," in *Proc. Design Automation Conf.*, pp. 703-708, June 1997.

[28] L. A. Belady, "A study of replacement algorithms for virtual storage computers," *IBM Sys. J.*, vol. 5, pp. 78-101, 1966.

[29] N. Young, "The k-server dual and loose competitiveness for paging," *Algorithmica*, vol. 11, pp. 525-541, June 1994.

[30] R. P. Dick, D. R. Rhodes, and W. H. Wolf, "TGFF: Task graphs for free," in *Proc. Int. Workshop HW/SW Co-Design*, pp. 97-101, Mar. 1998.