Term Paper

Hardware/Software Partitioning and Scheduling Algorithms for Dynamically Reconfigurable Architectures

Submitted by Mustafa Imran Ali
Fall 2003

Abstract

The use of dynamically reconfigurable logic (DRL) in the design of embedded systems has added a new dimension to the hardware/software co-synthesis problem. The spatial computing advantage and silicon reuse offered by DRL comes at the cost of reconfiguration latency which has to be effectively managed in order to optimize the system performance and power consumption. The initial attempts at leveraging DRL only aimed at optimizing the system performance, while recently power consumption minimization is also being researched. Another dimension is the dynamic run-time scheduling of DRL for applications having intrinsically dynamic behavior, as opposed to static scheduling explored by majority of the approaches. Nevertheless, the results obtained by applying these approaches show that the use of DRL promises reduction in system costs and increase in performance as compared to solutions which do not use DRL.

1. INTRODUCTION

The most widely accepted notion of an embedded system is a computer system that uses programmable processing unit(s) to implement part of its functionality but is not used as a general purpose computer system by the end user [1]. Examples of embedded systems range from

- simple appliances such as a microwave oven, where the embedded system provides a friendly interface,
- appliances for a computationally intensive task such as digital audio, video and graphics processing,
- hand-held devices, such as a cellular phone or PDA for which power consumption and size are critical but sophisticated tasks have to be performed,
- an industrial controller for which reliability, maintainability and ease of programmability are often concerns,
- safety-critical controller, such as an antilock brake controller in a car or an autopilot which must be able to meet hard real-time deadlines [2].

Although, embedded systems are not used as a general purpose computer system, multifunction systems that can switch among different functions are also included in this category. Since embedded computing systems usually form part of larger systems and due to the fact that they are designed for some specific purpose, the overall cost is the main driving factor in the design of these systems besides the system performance. With the emergence of battery operated devices, system power reduction has emerged as an objective competing with cost and performance goals.

The functionality of embedded systems is generally implemented with programmable CPU(s) running software implementing part of the functionality while the other performance critical portions are often implemented in hardware and/or, using reconfigurable logic. The emergence of systems-on-chip (SoC) paradigm, fueled by a desire to implement ever-increasing functionality into a single chip, encompasses embedded systems as well. For such complex systems manual design techniques fail to achieve the goals because of the large number of components involved and their complex interactions. There is strong need to explore the design space efficiently in order to meet the goals of cost, performance and power consumption. In order to consider the tradeoffs, there is a need to design the system taking into account the interaction between hardware and software components. This has led to the field of hardware-software co-design.

The co-design of embedded systems involves:

- Co-specification: Creating specifications that describe both hardware and software elements (and the relationship between them);
- Co-synthesis: Automatic or semi-automatic design of hardware and software to meet a specification;
- Co-Simulation: Simultaneous simulation of hardware and software elements, often at different levels of abstraction.

This paper focuses on the co-synthesis problem which involves four basic steps:

1. Partitioning the functional description into processes,
2. Allocating processes to processing elements,
3. Scheduling processes on the PEs,
4. Binding processing elements to particular component types

These phases are interrelated, but these are often separated to make the problem more tractable. As stated earlier, the partition of a system into hardware and software is of critical importance because it has a first order impact on the cost/performance characteristics of the final design. Allocation and binding are closely related and they involve the determination of the number of processing elements and the assigning of tasks to specific instances of PEs. Scheduling determines the times at which functions are executed, which is important when several functional partitions share one hardware unit.

Until recently, the widely used architectural template of embedded systems for hardware software partitioning was based on one or more programmable general purpose and/or special purpose (signal processing) processor(s), ASICs and memories hierarchy along with communication channels that connect the CPUs, ASICs and the memory through some communication topology as shown in **Figure 1.1**. With the emergence of commercial reconfigurable logic devices such as FPGAs,

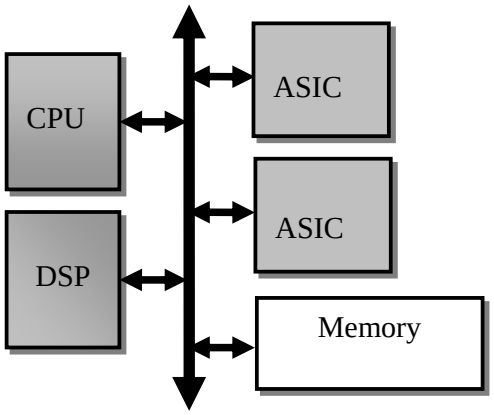

Figure 1.1: An architectural template for hardware/software partitioning

reconfigurable computing is emerging as an alternative to conventional ASICs and general-purpose processors. Reconfigurable architectures can be post-fabrication customized for a wide class of applications, including multimedia, communications, networking, graphics and cryptography, to achieve significantly higher performance over general purpose or even special purpose processor alternatives (such as DSPs and ASIPs). With recent advancements in the architecture of reconfigurable devices, dynamically reconfigurable devices have been proposed that can be partially reconfigured at run-time to implement different tasks without effecting computation of other tasks residing on the same device. Various research groups have demonstrated that a tightly coupled reconfigurable co-processor with a general purpose CPU can achieve significant speedup on a general class of applications. An abstract model of this new class of architectures is shown in **Figure 1.2**.

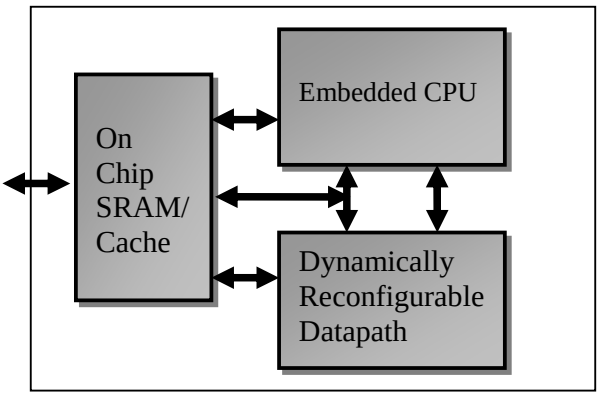

Figure 1.2: A template for an embedded system using DRL

The architecture extends the template presented earlier for embedded systems by replacing (and sometimes complementing) the ASICs. The CPU can be used to implement control-intensive functions and the system I/O, leaving the datapath to accelerate computation intensive parts of an application. This class of architectures defines a common, reusable platform for a wide range of applications, and potentially provides a better transistor utilization than a single CPU or combined CPU and ASIC of comparable silicon area. It also reduces the NRE costs and long design cycles associated with the development of ASICs.

This paper deals with the specific problem of hardware/software partitioning and scheduling in context of embedded systems design using the architectural template incorporating a dynamically reconfigurable logic (DRL) fabric. There has been considerable research effort in co-design of conventional embedded hardware/software architectures containing ASICs. However, the partitioning problem for architectures containing DRL has a different requirement: it demands a two-dimensional partitioning strategy, in both spatial and temporal domains, while the conventional partitioning involves only the spatial partitioning. Here, spatial partitioning refers to physical implementation of different functionality within different areas of the hardware resource. For dynamically reconfigurable architectures, besides spatial partitioning, the partitioning algorithm needs to perform temporal partitioning, meaning that the DRL can be reconfigured at various phases of program execution to implement different functionality. This paper is a survey of the various approaches proposed till date to address this area in embedded systems design.

The organization of the paper is as follows: section 2 gives introduces the problem formally and briefly explains the common concepts that apply to the hardware/software co-synthesis of .dynamically reconfigurable architectures. Section 3 presents the literature survey of the proposed approaches. Section 4 evaluates the various approaches and summarizes, and is followed by conclusion and bibliography.

2. PRELIMINARIES

2.1 Input Specification

For most co-synthesis algorithms, the input is assumed to be in the form of task-graphs. A task is a (atomic) portion of the computation an embedded system is required to carry out. Correlation function and convolution operation are examples of task types. Multiplication is also a task type, although past work typically assumes coarse-grained tasks, i.e, each task is assumed to be something that would require multiple instructions on a general-purpose processor (e.g., loops are examples of tasks). A task contains both data and control flow information. For some algorithms, the basic blocks (tasks) can be of any granularity, provided that external control transfers take place only at the end of a basic block. At the lowest level, a basic block is comprised of primitive operations such as additions, multiplications, memory loads, etc, with (un)conditional branches allowed only at the end of the block. At the highest level, basic blocks are functions/procedures. An embedded system specification may contain more than one task of the same type.

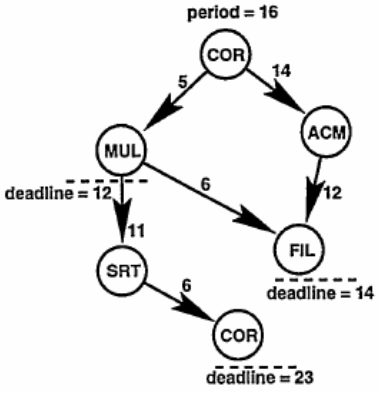

Figure 2.1: A task-graph

A task graph is a directed, acyclic, connected graph consisting of a collection of tasks each of which is associated with a task type, and a collection of directed edges, each of which is associated with a scalar denoting the amount of data which must be transferred between the tasks it connects. Task graphs are required to be acyclic to reduce the complexity of the co-synthesis problem. Edges represent communication events. In **Figure 2.1**, each circular node, denoting a task, is labeled with its task type and each directed edge is labeled with the amount of data which flows along it. Each edge

points away from its parent task and toward its child task. A task's parents are the tasks to which it is connected by incoming edges. A task's children are the tasks to which it is connected by outgoing edges. A directed edge may begin executing only after its parent task has completed executing. A task may begin executing only after all its incoming edges have completed executing. All tasks without outgoing edges have deadlines. However, any other task may also have a deadline (indicated by dashed lines in **Figure 2.1**). The task with no incoming edges is the start task. If a task does not complete its execution before its deadline is reached, hard real-time constraints are violated.

A task graph's period is the interval at which it repeats execution. It is possible for a task graph's period to be less than some of the deadlines of tasks within it. In embedded system specifications which contain such task graphs, the execution of multiple instances of the same task graph overlap in time. An embedded system specification may contain multiple task graphs, each of which may contain different tasks and deadlines. In addition, different task graphs may have different periods. Concurrent and multi-function embedded systems are modeled as a set of several task graphs.

In embedded system design which use events in the specification, when a task is ready for execution this is indicated explicitly by an event. An event consists of the following information: TaskId, TaskGraphId, TaskPriority and TaskType. Events may be sequentially ordered using the TaskPriority field. The list of sorted events is the Events Stream.

Multi-rate embedded systems consist of multiple periodic task graphs which may have different periods. The hyperperiod of a system is the least common multiple of the periods of the task-graphs in the system. A multi-rate schedule is valid if and only if all deadlines are met and each task graph is repeatedly executed until the hyperperiod has been reached.

A co-synthesis problem can be summarized as follows: Given embedded systems 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 Architecture Models

The term processing element (PE) is generally used to denote any device that executes tasks. Some authors use the term PPE to denote any programmable processing element such as processors and FPGAs. Most co-synthesis algorithms that target embedded systems with dynamically reconfigurable logic assume a target architecture that includes three key components: a software unit (a microprocessor), a dynamic hardware unit (a reprogrammable device), and a shared memory unit (communication link between software and hardware). The software can directly configure the hardware, which is partially reconfigurable. Partial reconfiguration allows for a selective change of hardware segments of arbitrary size at an arbitrary location, without disrupting the operation of the rest of the hardware space. Some early co-synthesis systems do not allow multiple tasks to execute concurrently on the same FPGA, while some use the simplifying assumption that the embedded system consists of just one processor and one FPGA. In most general systems, no limitation is placed on the quantity of system resources.

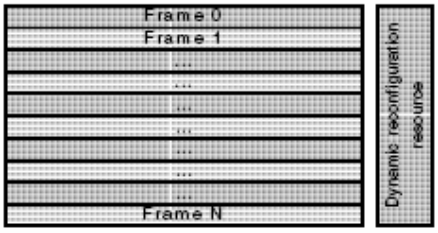

Figure 2.2: A one-dimensional reconfiguration model of a dynamically reconfigurable FPGA

Some co-synthesis systems assume a one-dimensional reconfiguration model for dynamically reconfigurable FPGAs as shown in **Figure 2.2**. 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.

2.3 Resource Libraries

Embedded system specifications are mapped to elements of a *resource library*, which consists of a PE library, a link library and memories. The PE library consists of various types of FPGAs, CPLDs, ASICs, and general-purpose CPUs. 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. There may be different hardware implementations of the same task each having a different execution time, CLB requirement and power consumption. Each ASIC is characterized by the number of gates, the number of pins, and price. Each general-purpose processor is characterized by the memory hierarchy information, communication processor/port characteristics, the context switch time, and price. Each pair of processor and tasks, there is a worst-case execution time, a preemption time, average power consumption and a memory load. Execution time is the amount of time a processor requires to carry out a task. Preemption time is the amount of time required to save a task's context before interrupting it with another task. Memory load is the amount of memory required by a task when executed on a processor. This variable accounts for instruction and data space.

The link library consists of various types of links such as point-to-point, bus, LAN. Each communication link is 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.

2.4 Profiling and Estimation

The co-synthesis algorithm performs the hardware/software partitioning and scheduling using various estimates such as worst-case execution time, communication-delays, average power consumption,

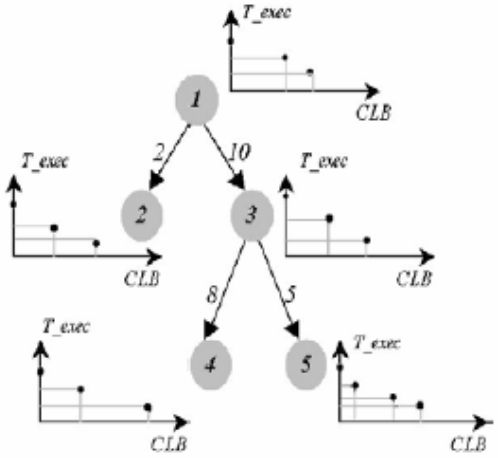

Figure 2.3: A task-graph and area-time tradeoff curves for task implementations

hardware implementation area etc. These estimates are obtained using various tools such as code profilers and high-level synthesis tools. **Figure 2.3** shows a task-graph with area-time

implementation points for different implementations of each task in hardware exploiting different amount of parallelism. The 0 CLB implementations are software implementations. This information is obtained during the profiling stage.

2.5 Partitioning and Scheduling

HW/SW partitioning and scheduling techniques can be differentiated in several ways. Partitioning can be classified as fine-grained (if it partitions the system specification at the basic-block level) or as coarse-grained (if system specification is partitioned at the process or task level). Also, HW/SW scheduling can be classified as static or dynamic. A scheduling policy is said to be static when tasks are executed in a fixed order determined offline, and dynamic when the order of execution is decided online. HW/SW tasks' sequence can change dynamically in complex embedded systems (i.e., control-dominated applications), since such systems often have to operate under many different conditions.

2.6 Issues/Challenges

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. A fourth sub-problem recently being addressed is that of power overhead due to the dynamic reconfiguration, which can account for half of the FPGA power consumption.

3. A SURVEY OF PROPOSED APPROACHES

3.1 STATIC SCHEDULING APPROACHES WITH COARSE-GRAINED PARTITIONING

3.1.1 CORDS: Co-synthesis of Reconfigurable Real-Time Distributed Embedded Systems[3]

CORDS was the first co-synthesis system that considered the effects of dynamically reconfiguring FPGAs during the operation of an embedded system, and reduce the amount of FPGA reconfiguration time. CORDS uses a preemptive, dynamic priority, multi-rate scheduling algorithm to deal with the problem of reconfiguration latency minimization. The main idea is to derive a schedule which locates different instances of the same task type adjacent to each other so that the number of reconfigurations an FPGA needs to undergo will be reduced, resulting in significant time savings.

CORDS automatically selects an allocation from a set of FPGAs, general-purpose processors, and communication resources. It assigns tasks to FPGAs and general-purpose processors, and determines the connectivity of communication resources. An evolutionary algorithm based on similar to a genetic algorithm is used to carry out the allocation optimization. Then the scheduling algorithm is invoked to determine the time at which each task is executed, the communication resource to which each communication event is assigned, and the time at which each communication event occurs. As this problem is NP-complete, CORDS uses a heuristic scheduling algorithm: a preemptive, static, critical-path scheduling algorithm with dynamic task reordering based on FPGA reconfiguration time. A static, critical path based metric, called slack is used to produce a preliminary order for tasks. A task's slack is the amount of time its execution can be delayed, from its earliest possible execution time, without causing any other tasks to miss their deadlines. When the scheduling algorithm begins, all start tasks are entered into a pending list which is sorted in order of decreasing slack. Tasks are sequentially removed from the end of the pending list and scheduled. After a task is scheduled, its children are checked to determine whether all of their parents have been scheduled, satisfying data

dependencies. Children which satisfy this test are entered into the pending list, reconfiguration delays are recalculated, and the pending list is sorted again before scheduling the next task.

There is a reconfiguration delay associated with every task which is assigned to an FPGA except for the reconfiguration delay for a task of type h, assigned to an FPGA whose most recently scheduled task was also of type h, is zero. Reconfiguration delay is dynamically adjusted during the execution of the scheduling algorithm. Every time a task is removed from the pending list, a dynamic check is first made to determine whether or not executing another task fist would be likely to reduce total FPGA reconfiguration time without causing deadlines to be missed. Dynamic priority is defined to be the sum of a task's negative slack and its negative reconfiguration delay, i.e.

$$dynamic\_priority = -slack - reconfiguration\_delay$$

If a task u, which was just removed from the pending list, is assigned to an FPGA then the dynamic priorities of all the other tasks in the pending list which are assigned to the same FPGA as u are compared with task u's dynamic priority. If another task has a higher dynamic priority than u, it is removed from the pending list and scheduled immediately, after which time u is again considered for scheduling.

When a task is scheduled on a processor, CORDS determines whether or not preemption is likely to result in an improved schedule.

The experimental results produced by CORDS indicated that time multiplexing tasks on dynamically reconfigurable FPGAs has the potential to decrease system price and allow solutions to specifications which are infeasible by using the processors alone.

3.1.2 CRUSADE: Hardware/Software Co-synthesis of Dynamically Reconfigurable Heterogeneous Real-Time Distributed Embedded Systems [4]

CRUSADE is a heuristic-based constructive co-synthesis algorithm which optimizes the cost of hardware architecture while meeting the real-time and other constraints. CRUSADE approaches the challenges of using dynamically reconfigurable logic as follows:

a. Non-overlapping task graphs offer opportunity to share the DRL and realize cost-effective architectures. Non-overlapping task graphs are identified by using a compatibility vector for each task graph (*Ti*) = [D*i1*, D*i2*, ..., D*ik*] that indicates compatibility of task graph *Ti* with other task graphs of embedded system. D*ij* indicates compatibility of task graph *Ti* with task graph *Tj*. If D*ij* = 0, it implies that task graph *Ti* is compatible with task graph *Tj* and 1, if otherwise. If execution times of two task graphs do not overlap, they are said to be *compatible* task graphs and they can share the DRL resources. If two task graphs are not compatible, it implies that their respective execution times indeed overlap and therefore an independent set of DRL resources must be assigned. 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 3.1**.

Task graphs for which compatibility vector is not specified, the architecture is built without requiring dynamic reconfiguration of the PPEs. Once the architecture is defined and deadlines are met, *merge potential* of the architecture is identified as summation of number of PPEs and links in the architecture. A Merge array is created 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. Each tuple is picked and its merge is explored by creating multiple modes for PPE and followed with scheduling and finish time estimation. If deadlines are met, the merge is accepted and the modified architecture will be used, otherwise it is rejected and the next merge from the merge array is explored. Once all merges have been explored, the modified architecture is compared with the previous architecture, and if the architecture cost or merge potential is decreasing, the process is repeated. The process stops when the architecture cost or merge potential can no longer be reduced.

```
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_i {
            merge_array = NULL;
            inter_PPE_tuple (PPE_i, PPE_k) ← group PPE_i
            with one of the adjacent PPE, PPE_k, with
            which the merge potential is maximum;
            add inter_PPE_tuple (PPE_i, PPE_k) to merge_array;
            PPE_i_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 3.1: The procedure for dynamic reconfiguration

b. During allocation of non-overlapping task sets to DRL, reconfiguration is taken into account. The non-overlapping task graphs are allocated using an allocation array which is an array of possible allocations at that point in co-synthesis. In the allocation array, multiple versions of each programmable device are provided. Each version corresponds to a 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, the various versions of the device are merged ensuring that real-time constraints are met.

c. Switching time between modes of a device, called boot time, is taken into consideration by adding a task *reboot_task* at beginning of each mode of a device, while checking whether the deadlines will be met.

d. The reconfiguration controller interfaces are modeled by creating a reconfiguration option array for various possible device programming options, each characterized by a dollar cost and a boot time and the array is ordered on the basis of increasing dollar cost. The lowest cost architecture is then chosen while meeting boot time requirements of the system.

e. To manage the delay constraints of the synthesized hardware logic blocks, two parameters called effective resource utilization factor (ERUF) and effective pin utilization factor (EPUF) are used. Their value is experimentally determined to guarantee the meeting of delay constraints during co-synthesis.

The steps in the CRUSADE flow include pre-processing, synthesis, and dynamic reconfiguration generation.

Pre-processing: Applies various transformations on the task-graphs, mainly clustering of tasks to reduce the search space for the allocation phase (the next step).

Synthesis: Each cluster is allocated in two steps: an outer loop for allocating each cluster and an inner loop for evaluating various allocations for each cluster.

Scheduling: The current allocation is scheduled to determine the relative ordering of tasks and their start and finish times while ensuring that deadlines are met.
Allocation evaluation: It compares the current allocation against previous ones based on total dollar cost of the architecture. If deadlines are met, the merging of different modes of a programmable device is explored.

The experimental results using the CRUSADE algorithm shows that significant cost savings are realized by using dynamic reconfiguration of programmable devices with a number large real-life field examples as compared to without it.

3.1.3 Hardware/Software Partitioning for Dynamic Reconfiguration using a Genetic Algorithm [5]

The approach in [5] is based on using a genetic algorithm to realize design space exploration by generating different mappings of the tasks on the processor and the FPGA, similar to the CORDS (Section 3.1.1) approach. Evaluation of the execution time of the architecture for each mapping requires defining a schedule of the tasks including reconfigurations for context switching and data transfers between tasks. This evaluation is performed by using a clustering heuristic similar to one used by approach used in Section 3.1.2. The task-graph is clustered and each tasks cluster comprises a single context for reconfiguration. The global execution time for a given tasks assignment after clustering the tasks determines the cost of the given solution which is to be optimized. Hence, the hardware/software partitioning is performed using the genetic algorithm while the clustering process groups tasks assigned to HW into contexts which are then scheduled based on the dependencies.

3.1.4 Hardware Software Co-synthesis for Run-Time Incrementally Reconfigurable FPGAs [6]

This approach aims to reduce the run-time overhead of reconfiguration by using concepts of Early Partial Reconfiguration (EPR) that minimizes the overhead by performing reconfiguration for an operation (a task) mapped to an FPGA as early as possible so that the operation is ready to start when its execution is requested. For further reduction of the overhead, the incremental reconfiguration (IR) of FPGAs is used with the EPR concept.

This EPR strategy saves much time compared to the lazy reconfiguration that starts reconfiguration of an operation when its execution is requested. The notion of pre-fetching the configuration to the target architecture is used, and it is assumed that the target architecture is composed of a CPU and a reconfigurable FPGA, where multiple computations can run concurrently on the FPGA along with the computation of CPU. It is also assumed that there is only one reconfiguration controller that performs reconfiguration jobs. The concepts of lazy reconfiguration and EPR are shown in **Figure 3.2**. **Figure 3.2(d)** shows how the reconfiguration jobs and their corresponding computation jobs are separated, to allow another reconfiguration job to be performed utilizing the remaining slack interval.

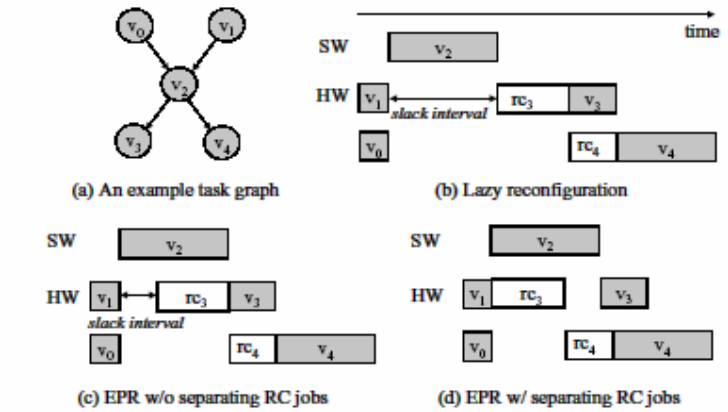

Figure 3.2: Early partial reconfiguration and splitting reconfiguration jobs

With incremental reconfiguration, the amount of reconfiguration data required during the reconfiguration process can be reduced. CORDS (Section 3.1.1) present a scheduling method which tries to schedule the tasks of the same type consecutively thereby eliminating the reconfiguration of the successor tasks of the same type. The IR approach further reduces the reconfiguration time overhead by performing incremental reconfiguration of tasks which partially share configuration data with tasks that have already been configured.

```
1.  currP = bestP = InitialP();
2.  IterationLoop: loop
3.      everbestP = bestP;
4.      while (UnlockedTaskExist) loop
5.          moved = SelectNextMove();
6.          currP = MoveAndLockTask(moved);
7.          bestP = GetBetterPart(currP,bestP);
8.      end loop;
9.      if no improvement of T^G in this pass then
10.         return everbestP;
11.     else // Do another pass
12.         UnlockAllTasks();
13. end loop;

14. SelectNextMove() {
15.     best_move = 0; best_TG = max number;
16.     for task v_i in all tasks
17.         TryMove(v_i);
18.         TG = GetScheduleLength();
19.         if( best_TG > TG )
20.             best_move = v_i; best_TG = TG;
21.         RestoreMove(v_i);
22.     end for;
23.     return best_move; }

24. GetScheduleLength() {
25.     CalculateECST();
26.     while (UnscheduledTaskExist) loop
27.         selected = SelectATask();
28.         ScheduleATask(selected);
29.         UpdateECST();
30.     end loop;
31.     return T^G; }
```

Figure 3.3: A pseudo code of the proposed HW-SW co-synthesis heuristic

The proposed heuristic is shown in **Figure 3.3**. The hardware/software partitioning is based on Fiduccia/Mattheyses algorithm and the concepts of EPR and IR are applied in the scheduling stage (GetScheduleLength() line 24) of the heuristic. The scheduler uses a list scheduling based on the earliest computation start times (ECST's) of ready tasks. The task that can start its computation at the earliest time, i.e. that has the smallest ECST among ready tasks is selected (SelectATask()) and its computation job (and reconfiguration job first, if it is mapped to HW) is scheduled (ScheduleATask(selected)).

To exploit the EPR and IR concepts, a present HW configuration set (PCS^{HW}) is defined as the set of HW tasks that currently occupy the HW resource. For example, in **Figure 3.4(a)**, since tasks v0 and v1 (in the task graph of **Figure 3.2(a)**) are mapped to HW, so PCS^{HW} = {v0, v1}. In **Figure 3.4**, the height of each rectangle represents the size, i.e. the implementation costs of each HW task.

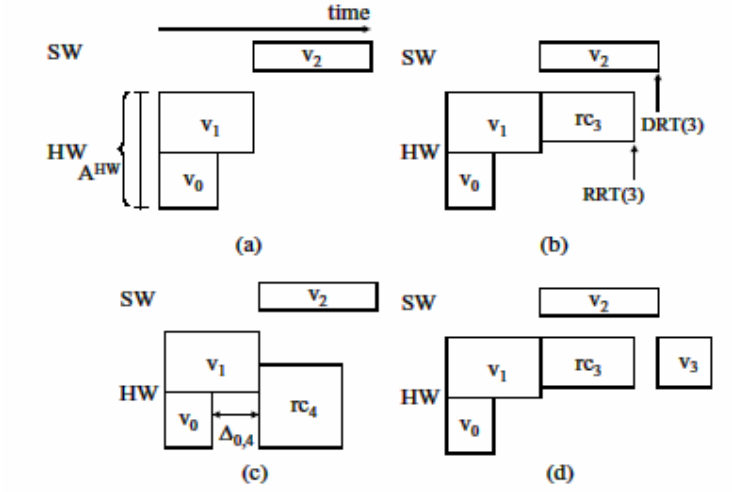

Figure 3.4: Examples of Scheduling HW tasks

The ECST of task vi is determined by the maximum of the time when data from its direct predecessors are ready to be used, known as the data ready time DRT(i), and the time when the computation resource (HW or SW) is ready to be used, known as the resource ready time RRT(i). For HW task vi, RRT(i) is the time point when the reconfiguration of HW task vi is completed. Note that when DRT(i) >= RRT(i), the reconfiguration overhead of HW task vi is totally hidden by EPR as shown in **Figure 3.4(b)**.

For a task vi, DRT(i) is determined as follows:

$$DRT(i) = \max_{v_j \in Pred(i)} \{T_F(j) + comm(j,i)\}$$

where, Pred(i) is the set of direct predecessor tasks of task vi and comm(j,i) is the communication time between task vj and vi. Since the predecessor task vj is already scheduled, its finish time T_F (j) can be obtained.

If task vi is mapped to SW, RRT(i) is determined to be the maximum of finish times of tasks scheduled on SW before task vi. If task vi is mapped to HW, PCS^{HW} should be considered in the computation of RRT(i) as follows:

$$RRT(i) = \min_{v_j \in PCS^{HW}} \{T_F(j) + (1 - \rho(j,i)) \cdot rc(i) + \Delta j,i\}$$

where, rc(i) is the reconfiguration time of task vi, $\rho(j,i)$ is the percentage of shared configuration data between tasks vj and vi from the viewpoint of task vi. In the equation, the term $1 - \rho(j,i) \cdot rc(i)$ implies the reconfiguration time overhead when task vi is incrementally reconfigured utilizing the configuration of task vj. $\Delta_{j,i}$ is exemplified in **Figure 3.4(c)**. In the Figure, if task v4 shares more configuration data with task v0 than with task v1, we can try reconfiguring task v4 just after the computation of task v0 finishes. However, in this case, since the sum of the size of task v4 and that of task v1 exceeds the given HW cost constraint A^{HW}. Thus, the reconfiguration of task v4 is delayed by the amount of $\Delta_{0,4}$ until task v1 releases the HW resource. For another example, in the case of **Figure 3.4(b)**, since the size of task v3 is smaller than that of task v1, $\Delta_{1,3} = 0$.

As shown in the above examples, to determine $\Delta_{j,i}$, we should consider two cases. Case I (II) represents a case where task vi shares configuration data with task vj $\in PCS^{HW}$ and the HW cost of task vj is larger than or equal to (smaller than) that of task vi. In Case I, $\Delta_{j,i} = 0$. In Case II, $\Delta_{j,i}$ is calculated as the time interval for which we have to wait to obtain enough HW area for reconfiguring task vi. **Figure 3.4 (d)** depicts the result of scheduling the reconfiguration and computation jobs of task v3.

The results of applying the proposed heuristic show that for a given hardware resource, significant performance improvements are obtained by using the concepts of EPR and IR for synthetic examples as well as real embedded system examples compared to the optimal solutions without these concepts.

3.2 FINE-GRAINED PARTITIONING

3.2.1 The NIMBLE Compiler [7]

The NIMBLE compiler incorporates a hardware/software partitioning algorithm that performs fine-grained partitioning at the loop and basic block levels and optimizes the global application execution times, communication time and datapath reconfiguration time. The input to the algorithm is a set of candidate loops for hardware, termed kernels, which have been extracted from the source application. Each loop has a software version and one or more hardware versions that represent different delay and area tradeoffs. The partitioning algorithm selects which loops to implement in the FPGA, and which hardware version of each loop should be used to achieve the highest application-level performance. Although partitioning is generally done at the loop-level, the partitioner can make basic-block level decisions by putting only a subset of the basic blocks of a kernel CFG into the hardware. **Figure 3.5(b)** shows an example of a loop where an infrequently executed branch has been trimmed by implementing it in software.

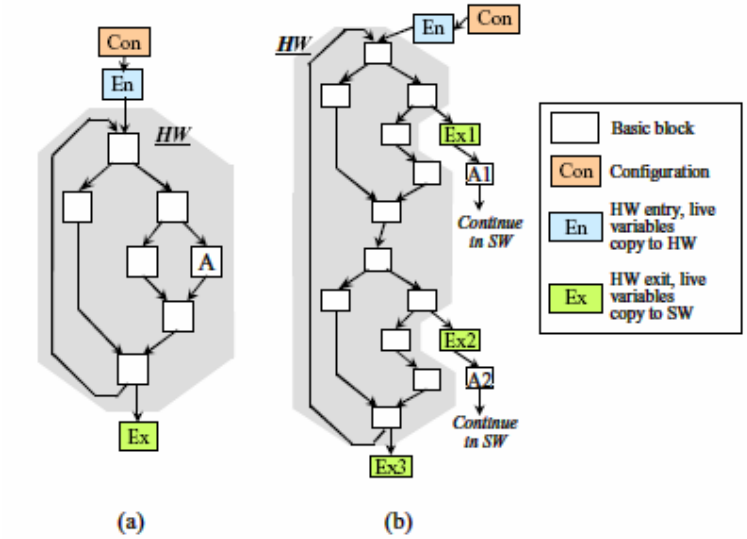

Figure 3.5: Multiple hardware kernels for one loop

As the algorithm tries to maximize the overall application performance, it uses a global cost function that incorporates the hardware and software execution times, hardware kernel entry and exit delay, and hardware reconfiguration time. Equation 1 shows the global cost of all loops T_{all_loops}, which is the sum of time spent in each individual loop $T(L_i)$. $T(L_i)$ denotes the total time spent in loop L_i, including all its iterations and entries**.**

$$T_{all\_loops} = \sum_{i \in L} T(L_i) \text{.....} Eq(1)$$

$$T(L_i) = T_{SW}(L_i) \bullet Iter(L_i) \text{.....} Eq(2)$$

$$T(L_i, K_j) = T_{HW}(L_i, K_j) \bullet Iter(Li) + T_{SW}(L_i, K_j) \bullet Iter(Li) +$$
$$T_{SW2HW}(L_i, K_j) \bullet En(Li, K_j) + T_{HW2SW}(L_i, K_j) \bullet Ex(Li, K_j) +$$
$$T_{config}(L_i, K_j) .....Eq(3)$$
$$T_{config}(L_i, K_j) = N_{miss}(L_i) \bullet T_{miss}(L_i, K_j) + N_{hit}(L_i) \bullet T_{hit}(L_i, K_j) .....Eq(4)$$

Equation 2 is when Li is implemented in software and Equation 3 is for kernel Kj of Li implemented in hardware. As shown in Equation 2, if L_i is selected to be implemented in software only, its execution time can be characterized as the average time per iteration $T_{sw}(L_i)$ times its number of iterations $Iter(L_i)$. The computation of hardware time is more complex. Suppose we put kernel version K_j of loop L_i in hardware. The hardware loop time shown in Equation 3 is composed of several terms:

1. Execution time spent in the hardware itself. Similar to software time, it is the average hardware time per iteration $T_{hw}(Li, Kj)$ times the number of iterations $Iter(L_i)$.
2. Execution time spent in the software if kernel K_j only implements a portion of the loop in the FPGA. (See **Figure 3.5(b)** for an example of a partial loop in hardware.)
3. Communication time between hardware and software, which involves the copying of live variables to and from the FPGA. Since variable transfer only happens when the program enters orexits from hardware, it is obtained by multiplying the cost per transfer (T_{hw2sw} or T_{sw2hw}) and the number of hardware entries $En(Li, Kj)$ and exits $Ex(Li, Ki)$, respectively.
4. Configuration time of the loop on the FPGA. Unlike the previous terms which only depend on decisions made about the current loop L_i, configuration time depends on decisions made for other loops that interleave with L_i during application execution. Some architectures (such as the GARP[8]) utilize a configuration cache to store recent configurations, so that they can be quickly reconfigured. The configuration cost for the cache miss $(T_{miss}(Li, Kj))$ and hit $(T_{hit}(Li, Kj))$ can be dramatically different, therefore, they must be computed separately as shown in Equation 4. The numbers of configuration cache hits and misses $(N_{hit}(Li)$ and $N_{miss}(Li))$ for a loop depend what hardware/software partitioning decisions are made for all loops.

If configuration time is not included, optimizing execution time can be reduced to locally selecting the fastest implementation of each loop that satisfies the FPGA size constraint. However, because of the complexity of computing configuration cost, the partitioning problem is NP-complete, and involves evaluating loops in a global cost function to find the optimal solution.

Algorithm Flow: Since the total number of kernels can be large for many applications, a heuristic algorithm is deployed to efficiently solve the hardware/software partitioning problem. The two key heuristics applied are:

1. Reducing the number of loops and kernels that the algorithm needs to analyze, by focusing solely on "interesting" loops that contribute significantly to the application time.
2. For the remaining loops, partitioning them into small clusters and performing optimal selection in each loop cluster.

Based on the above heuristics, the partitioning algorithm consists of the following main steps:

1. Loop entry trace profiling (LEP). LEP generates a complete trace that records all loops entries, such that the configuration cost for all loops can be inferred.
2. Interesting loop detection (ILD). ILD screens all hardware candidate loops and only selects a subset of "interesting" loops.
3. Intra-loop kernel selection. This selects the best hardware kernel among the multiple versions of a loop implementation.
4. Inter-loop selection. Selects among loops and decides which go into hardware and software, respectively.

Steps 2 and 3 apply the first heuristic, in an attempt to cut down the number of loops and kernels to be considered. Step 4 applies the second heuristic and is the most critical step of the algorithm.

Inter-loop selection: This is the most critical step of the algorithm, as the final partitioning decision has to be based on the global cost function described earlier in Equation 1. Selections in previous steps eliminate loops/kernels based on execution time metrics for each individual loop, while in this step, the interaction among all loops is analyzed, and execution time and configuration time is optimized.

Even though the number of loops (say *n*) left after steps 1 and 2 may not be very large, the number of configuration possibilities is exponential (2^n). A clustering technique is therefore used to partition loops into small clusters to allow the partitioning problem to be solved optimally for each cluster.

Hierarchical Loop Clustering Based on the Loop-Procedure Hierarchy Graph: Clustering of loops is based on the loop-procedure hierarchy graph (LPHG) which represents the procedure call and loop nest relations in the application. **Figure 3.6** shows the LPHG for the wavelet image compression benchmark. A square node indicates a procedure definition, and a circular node indicates a loop. Edges into a procedure node represent calling instances to that procedure. An edge from a procedure to a loop indicates the loop is defined within the procedure. An edge from a loop *a* to another loop *b* indicates that *b* is nested inside loop *a*. There may be multiple incoming edges for a procedure, indicating multiple calling instances of the same procedure. Recursive procedures create cycles in the graph.

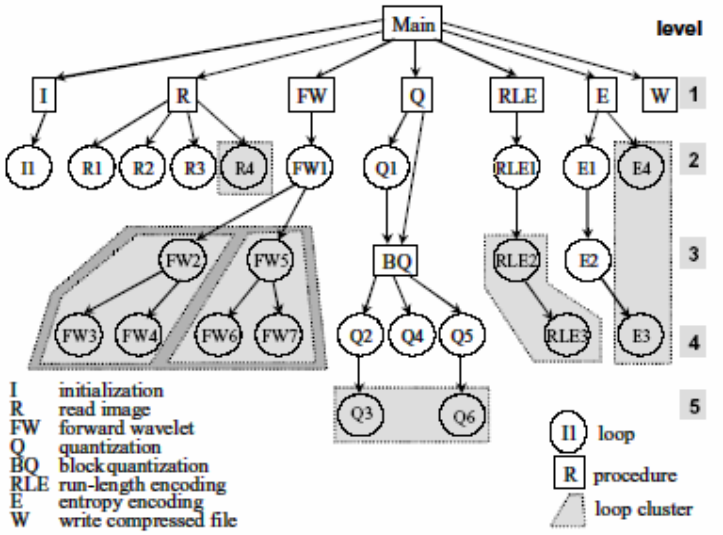

Figure 3.6: Loop-procedure hierarchy graph for a wavelet image compression benchmark

An LPHG captures loops and their relative positions in the application and therefore provides a navigation tool for the partitioning algorithm to traverse the loops. A shortest distance is defined from a node to the root node (*main*) as the *level* of that node. The following observations are made:

- If two loops have different first-level predecessors, they appear in a disjoint part of the LEP trace and do not compete for the FPGA configuration. For example, in **Figure 3.6**, all entries of loop FW3 appear strictly before those of RLE2. These loops can be partitioned into different clusters.
- Conversely, loops sharing common loop or procedure predecessors tend to compete with each other, and therefore should be placed in the same cluster. In the example, entries of FW3 and FW4 interleave and hence compete for the FPGA resource.

Based on the above observations, a hierarchical loop clustering algorithm based on the LPHG has been proposed. A predefined size limit for the loop clusters is used to ensure that the clusters are small enough for feasible optimal selection. The loop clustering algorithm traverses the loop-procedure hierarchy graph in a top-down fashion and recursively clusters loops until the sizes of all clusters are within the pre-defined limit. The algorithm works as follows:

1. Starting from the first level of the loop-procedure hierarchy, loops with a common predecessor at this level are clustered together. In **Figure 3.6**, the unshaded loop nodes are discarded after ILD step. Clusters {R4}, {FW2, FW3, FW4, FW5, FW6, FW7}, {Q3, Q6}, {RLE2, RLE3}, and {E4, E3} are generated based on their level 1 predecessors.
2. If the size of any cluster exceeds the cluster size limit, the graph is traversed down a level further in the hierarchy and the clusters are refined by grouping loops again with common predecessors at the new level. For example, the FW loop cluster has six loops. If the size limit is set at 5, there is a need to go down a level, to level 2, and recompute the clusters. All the other clusters are within the size limit and need no further refinement.
3. Repeat step 2 until all loop clusters satisfy the cluster size limit. In the example, at level 2, the FW loops still can not be resolved into smaller clusters and it is necessary to go down to level 3. The clustering result is shown in the figure, {FW2, FW3, FW4} and {FW5, FW6, FW7}.

Optimal Selection in Loop Clusters: After the loops have been partitioned into smaller clusters, the algorithm performs optimal hardware/software partitioning for each individual loop cluster. The approach adopted is to exhaustively search the solution space of all partitioning possibilities, evaluate each of these possibilities, and select the one with the best overall performance for all loops in the cluster. To evaluate the overall performance, the number of reconfigurations needed in each partitioning possibility must be computer. This is achieved by walking through the compressed loop entry trace. The state of the configuration cache (if there is one) is taken into account to estimate the number of hits and misses.

The NIMBLE hardware/software partitioning algorithm has been applied on real benchmarks. The Nimble flow takes off-the-shelf C code and compiles it onto a target architecture of a combined CPU and FPGA. The results indicate that near to optimal performance is obtained. While the algorithm consumes comparable CPU time to that of a greedy local-optimal algorithm, it generates close-to-optimal hardware/software partitions in all the benchmarks used.

3.3 GENERIC PARTITIONING WITH ENERGY-DELAY MINIMIZATION [9]

This approach in [9] targets the minimization of energy-delay costs associated with both computation and configuration. The addressed problems are energy-delay product minimization, delay-constrained energy minimization, and energy-constrained delay minimization. The problem is solved using network flow techniques, after transforming the original control flow graph into an equivalent network.

In the cost function used, the application delay is defined as a weighted sum of computation delays of all the basic blocks and the reconfiguration delays of all control transfers. The application energy is defined as a weighted sum of computation energies of all the basic blocks and the reconfiguration energies of all control transfers. The weights used are execution frequencies, which are derived from application profiling data. For the given task-graph, the cost of a node depends whether it is in hardware or in software, and the cost of an edge depends whether its origin is in software or in hardware and whether its destination node is in software or in hardware.

Each node *i* in the task graph is assigned a variable x_i=0, a cost c_0(i) and a weight w_0(i), if it is mapped to software, and a variable x_i=1, a cost c_1(i) and a weight w_1(i), if it mapped to hardware. Each edge (i,j) in the task graph is associated with one of the four costs and one of the four weights, respectively, depending on the mapping of the nodes i and j:

$$c_{00}(i,j) \quad and \quad w_{00}(i,j), \quad if \ \bar{x}_i \bar{x}_j = 1$$
$$c_{01}(i,j) \quad and \quad w_{01}(i,j), \quad if \ \bar{x}_i x_j = 1$$
$$c_{10}(i,j) \quad and \quad w_{10}(i,j), \quad if \ x_i \bar{x}_j = 1$$
$$c_{11}(i,j) \quad and \quad w_{11}(i,j), \quad if \ x_i x_j = 1$$

The cost/weight can be either the energy or the delay or the energy delay product of a node/edge. Two assumptions are made about the edge costs and weights. First, transferring control from a hardware block to a software block is more expensive than vice versa. This is realistic since a hardware-to-software transition may cause extra data traffic. Second, transferring control from a software block to a hardware block is more expensive than vice versa. This is also realistic since a hardware-to-hardware transition may be less expensive due to partial reconfigurability. These two assumptions are stated as

$$c_{10}(i,j) \geq c_{00}(i,j) \ and \ w_{10}(i,j) \geq w_{00}(i,j)$$
$$c_{01}(i,j) \geq c_{11}(i,j) \ and \ w_{01}(i,j) \geq w_{11}(i,j)$$

The objective function F and the constraint function G are defined respectively as

$$F = \sum_{i \in V} f(i) + \sum_{(i,j) \in E} f(i,j)$$
$$G = \sum_{i \in V} g(i) + \sum_{(i,j) \in E} g(i,j)$$

where

$$f(i) = x_i \, c_1(i) + \bar{x}_i c_0(i),$$
$$g(i) = x_i \, w_1(i) + \bar{x}_i w_0(i),$$
$$f(i,j) = x_i \, x_j c_{11}(i,j) + x_i \, \bar{x}_j c_{10}(i,j) + \bar{x}_i \, x_j c_{01}(i,j) + \bar{x}_i \, \bar{x}_j c_{00}(i,j),$$
$$g(i,j) = x_i \, x_j w_{11}(i,j) + x_i \, \bar{x}_j w_{10}(i,j) + \bar{x}_i \, x_j w_{01}(i,j) + \bar{x}_i \, \bar{x}_j w_{00}(i,j)$$

In the unconstrained hardware-software bipartitioning problem, the task is to find an assignment of each partitioning variable x_i, such that sum of costs over all nodes and over all edges (the cost F of a bipartition) is minimized. The energy-delay product minimization is the unconstrained bipartitioning problem, with the cost defined in terms of the energy-delay product. For this problem, the CFG with the nodes and edge costs defined as before is transformed into a network, so that a minimum cut in the network corresponds to an optimal bipartition of the CFG.

The constrained hardware-software bipartitioning problem requires finding an assignment of each partitioning variable x_i such that the objective function is minimized, and the sum of weights over all nodes and all edges (the weight G of a bipartition) does not exceed some given budget B. The delay-constrained energy minimization or energy-constrained delay minimization is the constrained bipartitioning problem with the cost and the weight defined in terms of energy or delay, accordingly. For the constrained versions, the goal is then to map each node to either hardware or software (thus, changing costs and weights of nodes and edges) so that the overall cost is minimized and the overall weight does not exceed the budget. Two iterative methods are proposed to find a good solution in polynomial time.

By using the proposed heuristic on a number of examples, near-to-optimal solutions (which were obtained from an enumerative exponential-time algorithm) are obtained for the hardware/software partitions.

3.4 MULTIOBJECTIVE CO-SYNTHESIS

3.4.1 SLOPES: Hardware-software co-synthesis of low power real-time distributed embedded systems with dynamically reconfigurable FPGAs [10]

This approach is an extension of the MOGAC [11] approach. An evolutionary algorithm is used to tackle the problem of allocation and assignment. Scheduling is then performed in the inner loop of co-synthesis. Since the dynamically reconfigurable FPGA scheduling problem includes both the time and space domains, the scheduler is a two-dimensional multi-rate cyclic scheduling heuristic that takes into account the delay and power overheads of dynamic reconfiguration. Depending on the resource and reconfiguration information, the scheduler treats each task fairly and tries to globally minimize the reconfiguration overhead. The proposed 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.

An overview of the co-synthesis system is shown in **Figure 3.7**. 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.

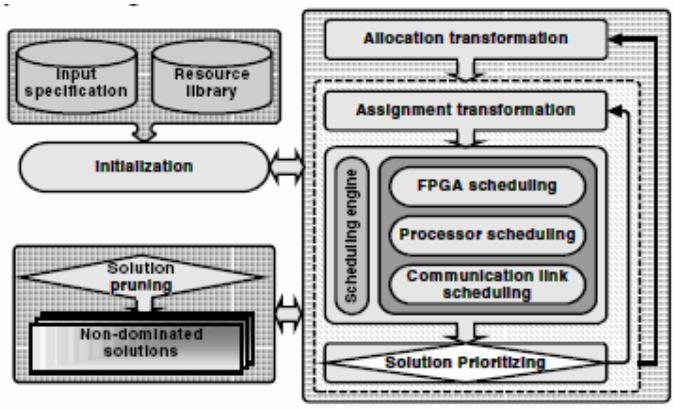

Figure 3.7: Hardware/Software co-synthesis overview

The Sechuling Algorithm: The proposed co-synthesis system assumes a one-dimensional FPGA reconfiguration model. Two issues have to be considered while scheduling:

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.

The approach taken handles these issues as follows:

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

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.

Details of these approaches follow.

Scheduling Sequence: A dynamic priority based approach is proposed, which dynamically updates the task priority instead of static slack-based priorities that are commonly used to order tasks for scheduling on processors. The intuitive idea behind using slack-based 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. 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. The dynamic priority based approach dynamically updates the task priority, as follows:

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

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,

$reconfig\_overhead_{task_i}$ is the reconfiguration overhead of $task_i$,

$reconfig\_inter_{task_{i,j}}$ 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.

Using this approach, both the real-time constraints and reconfiguration overhead are considered, and tasks from different task graphs are treated fairly.

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. All the frames assigned to a task need to be contiguous. 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:

$$eviction\_\cos t = \sum_{i=start\_frame}^{end\_frame} recurrent\_freq_{frame_i}$$

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.

The 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.

The Scheduling Algorithm: The pseudo-code for the two-dimensional scheduling algorithm is shown in **Figure 3.8**. 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 3.8: Pseudo-code of the scheduling algorithm

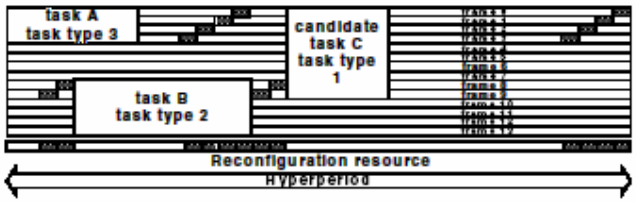

Figure 3.9: A task scheduling example

Consider task *C* in the partial FPGA schedule shown in **Figure 3.9**. When this task is being loaded into the FPGA, the reconfiguration overhead may be introduced before or after the task, shown as shaded blocks. Two issues need to be considered for the reconfiguration blocks introduced before task *C*. First, the time spans 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 the proposed approach, the priority, P_{frame_i}, to determine the reconfiguration sequence of frames is defined as follows:

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

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 non-increasing 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 3.9**, before candidate task *C*, frames 8 and 9 are scheduled first then frames 0 to 3.

The issues involved in scheduling reconfiguration slots after the task are as follows. 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:

$$P_{frame_i} = \begin{cases} -(-r\_t_{task} + f\_t_{frame_i}), & r\_t_{task} \le f\_t_{frame_i} \\ -(hyperperiod + f\_t_{frame_i} - r\_t_{task}), & r\_t_{task} > f\_t_{frame_i} \\ r\_t_{task} = ready\_time_{task} \bmod ulo \quad hyperperiod \\ f\_t_{frame_i} = finish\_time_{frame_i} \bmod ulo \quad hyperperiod \end{cases}$$

where

$finish\_time_{frame_i}$ is the finish time of the empty time-span 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 3.9**, 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 3.10**. In lines 3 to 6, the

```
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 3.10: Candidate position priority calculation

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 earlier 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 3.11**. The candidate position with the highest priority is chosen from *candidate_position_pool* (line 3). The

```
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 3.11: Task Scheduling

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 in order to try to schedule the task (line 11).

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

3.5 HW/SW PARTITIONING AND DYNAMIC RUN-TIME SCHEDULING [11, 12]

All the approaches discussed earlier are based on static scheduling. Embedded systems with dynamic behavior (e.g. MPEG-4), require run-time scheduling. Also, typical static scheduling algorithms assume that the task's execution time is the worst-case execution time (WCET). However, systems designed using WCET estimates could be highly under-utilized. The execution time of a task is rarely deterministic. For instance, it could be "data-dependent" (e.g., run-length encoding of video frames depends on the information within frames). Moreover, the execution time could depend on the available resources, especially when multiple applications share a system. Multi-function systems support multiple functions or applications of which only one is executed at any instant, depending on external factor. For these reasons, run-time scheduling is attractive and has been addressed recently.

The approach presented in [11] and [12] proposes a complete HW/SW system with a dynamically reconfigurable architecture along with a HW/SW co-design methodology for dynamic scheduling. The proposed system also takes power consumption optimization into consideration [12]. The proposed methodology addresses the problem of run-time HW/SW codesign for discrete event systems using an heterogeneous architecture that contains a standard off-the-shelf processor and a DRL based architecture. The proposed methodology follows an object orientation paradigm.

3.5.1. Definitions

Def. 1: a Discrete Event Class is a concurrent process type with a certain behavior, which is specified as a function of the state variables and input events.
Def. 2: a Discrete Event Object is a concrete instance of a DE class. Several DE objects from a single DE class are possible. Given two DE objects (DEO_1 and DEO_2) they may differ in the value of their state variables.
Def. 3: an Event E is a member of $T \times C \times O \times V$ where C is a given set of DE classes, O a set of DE objects, T a set of tags, $T \in \Re^+$ (the real numbers) and V a set of values. Tags are used to model time, and values represent operands or results of event computation.
Def. 4: an Event Stream (ES) is a list of events sequentially ordered by tag. Tags can represent, for example, event occurrence or event deadline.
Def. 5: Discrete Event Functional Unit is a physical component (i.e. DRL device or SW processor) where an event $e = (t, c_1, o_1, v_1)$ can be executed. A functional unit has an active pair (*class*, *object*), $p = (c_a, o_a)$. The methodology assumes that: (1) several DE classes could be mapped into a single DE

functional unit. (2) all DE objects from a DE class are mapped into the same DE functional unit where the DE class has been mapped.

Def. 6: an Object Switch is the mechanism that allows a DE functional unit to change from one DE object to another, both DE objects belonging to a same DE class. For example, if an input event e = (t, c_1, o_1, v_1) have to be processed in a DE functional unit with an active pair p = (c_1, o_2) then an object switch should be performed.

Def. 7: a Class Switch is the mechanism that allows a DE functional unit to change from one DE class to another. For example, if an input event e = (t, c_1, o_1, v_1) should be processed in a DE functional unit with an active pair p = (c_2, o_2), then a class switch should be performed. Class switch, in case of a DRL device, means a context reconfiguration. Object switch means to change the values of the state variables from the ones of a concrete DE object (o_1) to the others of another DE object (o_2).

3.5.2. Run-Time HW/SW Co-design Methodology

The proposed methodology is depicted in **Figure 3.12**. It is divided into three stages: *Application Stage*, *Static Stage* and *Dynamic Stage*. The key points in the methodology are: (1) *application* and *dynamic stages* handle DE classes and objects, and (2) *static stage* only handles DE classes.

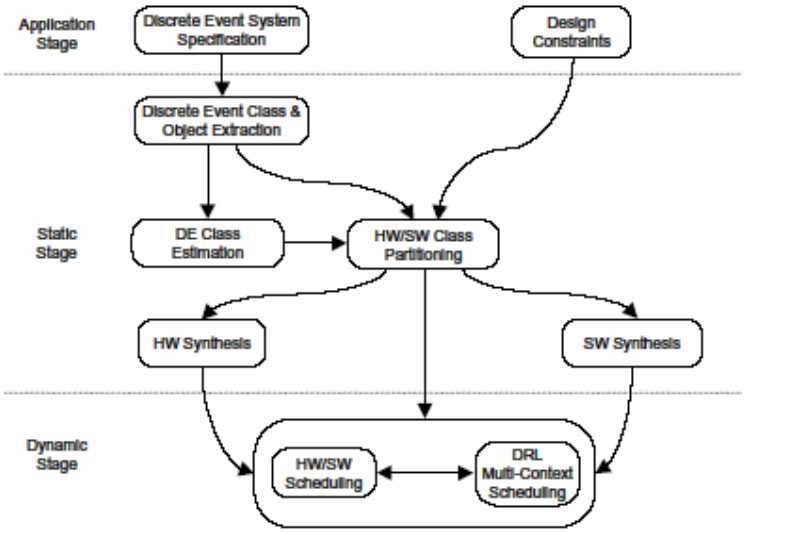

Figure 3.12: HW/SW Co-design Methodology

The *application stage* includes *Discrete Event System Specification* and *Design Constraints*. The use of a homogenous modeling language for system specification is assumed, where a set of independent DE classes must be firstly modeled. Afterwards, these DE classes are used to specify the entire system as a set of interrelated DE objects, which communicate among them using events. These DE objects are interrelated creating a concrete topology. A DE object computation is activated upon the arrival of an event. By design constraints we understand any design requirement necessary when synthesizing the design (i.e. timing or area requirements).

The *static stage* includes typical phases of a co-design methodology: (1) *estimation*, (2) *HW/SW partitioning*, (3) *HW and SW synthesis*, and (4) *extraction*.

As stated, the *static stage* handles DE classes and the system has been specified as a set of interrelated DE objects, which are instances of also specified DE classes. The final goals of the methodology's extraction phase are, for a given DE class, to obtain: (1) a list of all its instances (DE objects), and (2) a list of all different DE classes and objects connected to it. Both lists are afterwards attached to each DE class found in the system specification. Once this phase has finished, DE classes can be viewed as a set of independent processes or tasks.

The HW/SW partitioning approach is coarse-grained, since it works at the DE class level. Different HW/SW partitioning algorithms can be applied depending on the discrete event application to be solved. The solution given by the partitioning algorithm should meet design constraints. In Section 3.5.4, an example of HW/SW partitioning algorithm is proposed. It can be noted in the methodology, that although it addresses DRL architectures, a temporal partitioning phase is not present. The DE object/class extraction phase should be viewed as the temporal partitioning. Indeed, the temporal partitioning algorithm is included within the concept of DE class, because DE classes are functionally independent tasks.

The estimation phase also deals with DE classes, and used estimators depend on the application. Typically used estimators (HW/SW execution time, DRL area, etc) can be obtained using high-level synthesis and profiling tools.

The *dynamic stage* includes *HW/SW Scheduling* and *DRL Multi-Context Scheduling*. Both schedulers base their functionality on events present in the event stream. The methodology assumes that both of them are implemented in hardware using a centralized control scheme. As it is shown in **Figure 3.12**, these scheduling policies (HW/SW and DRL) co-operate and run in parallel during application run time execution, in order to meet system constraints (i.e. minimize the total application execution time parallelizing event executions with DRL reconfigurations). The aim of the HW/SW scheduler is to decide at runtime the execution order of the events stored in the event stream, in order to meet system constraints. Diverse policies could be implemented by the HW/SW scheduler based on the final application requirements (i.e. earliest deadline first using or not of a pre-emptive technique).

On the other hand, the DRL multi-context scheduler should be viewed as a tool used by the HW/SW scheduler in the sense that its goal is to facilitate or minimize the class switching mechanism to the HW/SW scheduler. It is assumed that different DRL schedulers can be defined depending on the application. In Section 3.5.5, a dynamic DRL multi-context scheduler is presented as an example.

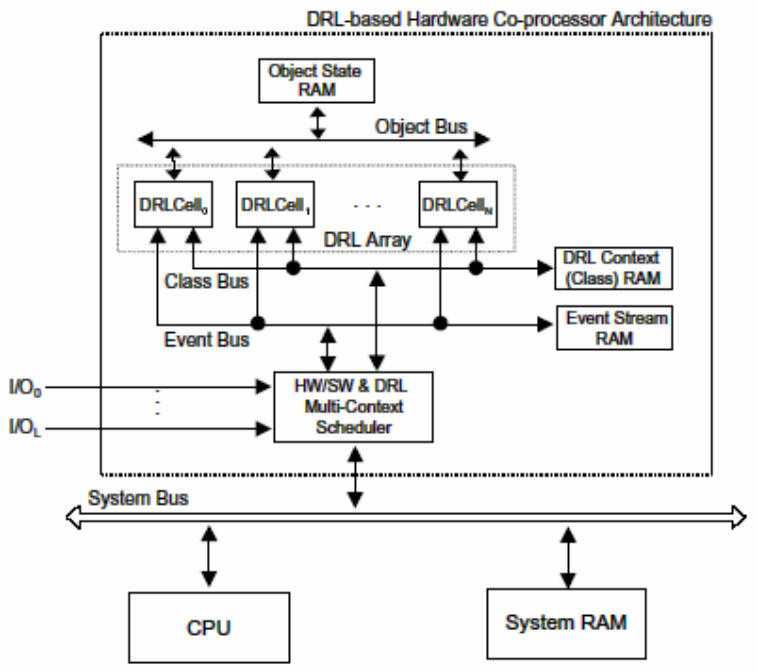

Figure 3.13: The target architecture

3.5.3. Target Architecture

The target architecture is depicted in **Figure 3.13**. It is an architecture which comprises a software processor, a DRL-based hardware architecture and shared memory resources. The software processor is a uni-processing system and it can execute only one event at a time. The DRL-based hardware co-processor can execute multiple events concurrently. Hardware and software co-operate (interact) via a DMA based memory sharing mechanism. The DRL-based hardware co-processor architecture is divided into: (1) HW/SW and DRL Multi-Context Scheduler, (2) DRL array, (3) Object State memory, (4) DRL Context memory and (5) Event Stream memory.

The HW/SW and DRL Multi-Context Scheduler must implement functions associated to the dynamic stage of methodology, as explained above. Events get the central scheduler through I/O ports or as a result of a previous event computation. The Event Stream is stored in the Event Stream memory. DRL contexts (which correspond to several DE classes from an application) are stored in the DRL Context memory. Finally, DE objects states are stored into the Object State memory.

The DRL array communicates with these memories and the central scheduler through several and functionally independent busses (Object, Class and Event busses). It is assumed that each DRL array element, named *DRL cell*, can implement any DE class with a required area ≈ 20K gates.

The proposed DRL co-processor architecture is scalable, and it is possible to implement any associative mapping between DE objects/classes and DRL cells. Note that this mapping is not only fixed by the structure of the DRL Context memory, but it also depends on the structure of the Object State memory.

3.5.4 HW/SW Partitioning Algorithm

This section presents a resources (object and class memory) constrained HW/SW partitioning algorithm as an example for this methodology.

Problem statement: Lets consider a set of independent DE classes $C = (C_1, C_2, ..., C_L)$, where each class, say C_i, is characterized by a set of estimators E_i,

$$E_i = (WCET_i^{HW}, WCET_i^{SW}, SVM_i, DRLA_i)$$

where
$WCET_i^{HW}$ stands for Worst Case Execution Time for a hardware implementation of the DE class C_i.
$WCET_i^{SW}$ stands for Worst Case Execution Time for a software implementation of the DE class C_i.
SVM_i stands for State Variables Memory size required by the class.
$DRLA_i$ stands for DE class DRL required Area.

Lets also consider the design constraints to be object memory and class (DRL context) memory constraints. That is, the total object state memory is denoted by OSMA (Object State Memory Available). CMA stands for the total amount of Class Memory Available.

We state our problem as maximizing the number of DE classes mapped to the DRL architecture while meeting memory resources constraints and DRL cell available area.

$$Max(|C^{HW}|), s.t. \sum_{j=1}^{M} SVM_j < OSMA, DRLA_j \leq CMA)$$

where
C^{HW} is the set of DE classes mapped to hardware, $C^{HW} = \{ C^{HW}_1, C^{HW}_2, \ldots, C^{HW}_3 \}$, $C^{HW} \subseteq C$

List-based HW/SW partitioning algorithm: The proposed HW/SW algorithm is a list-based partitioning algorithm. The algorithm maps more time consuming DE classes to hardware, in order to

minimize the total execution time at run-time, which will be responsibility of the HW/SW and DRL context scheduler. Thus, the set of input DE classes must be sequentially ordered and more time consuming DE classes should be prioritized when mapping to hardware. This objective is implemented using a cost function. For this example, the following cost function is proposed, although other cost functions could be applied.

$$F_i = \alpha \cdot (WCET_i^{HW} - WCET_i^{SW}) + \beta \cdot SVM_i$$

Indeed, this cost function prioresses DE classes with significant difference in its HW and SW execution times. Lower values, as result of applying this cost function, are assumed better than higher values. So, the sort function classifies values from lowest to highest. The pseudo-code of the proposed HW/SW partitioning algorithm is shown in **Figure 3.14**. It obtains the initial sequentially ordered list ($P_{INITIAL}$) after the cost function has been applied to all DE classes. Afterwards, the algorithm performs a loop, and tries to map as many DE classes to hardware as possible while memory and DRL area constraints are met. *AvailableResources()* function is responsible for design constraints checking. Mainly, it checks that the current hardware partition plus DE class C_i complies with design constraints.

```
ListBasedPartitioningAlgorithm(ED_Classes)
{
 PSW = { ø };  PHW = { ø };
 PINITIAL = Sort_DE_Classes_List (DE_Classes,
                                  FSORT);
 Ci = GetFirst(PINITIAL);
 for i = 2 to L loop
   if Ci.DRL_RequiredArea > DRL_Area then
      PSW = PSW  U  Get(Ci, PINITIAL);
   else
      if AvailableResources(Ci)) then
         PHW = PHW  U  Get(Ci, PINITIAL);
      else
         PSW = PSW  U  Get(Ci, PINITIAL);
      end if;
   end if;
 end loop;
}
```

Figure 3.14: List based partitioning algorithm

3.5.5: Run-time DRL multi-context scheduler: The presented scheduler assumes that the Event Stream is sorted. In this example, it is also assumed that only the first event of the event stream is been processed on a DE functional unit (DRL cell or CPU) at the same time. Modifications of this scheduler are possible in order to have several events being processed in parallel within the target architecture.

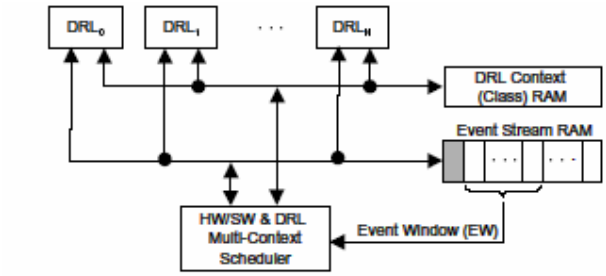

Figure 3.15: HW/SW and DRL multicontext scheduling

The key idea of the scheduler is to minimize class switching (DRL reconfiguration) overheads, in order to minimize the total application execution time. This objective is accomplished using a

lookahead strategy into the event stream memory (see **Figure 3.15**). Event Window (EW) describes the number of events that are observed in advance and is left as a parameter of our scheduler.

From the DRL array state (that is, from the DE classes that are active) and the event window, the DRL scheduler must decide which DE class should be removed (replaced) from the DRL array, and which DE must be loaded into.

Multi-context scheduling algorithm: The pseudo-code for the dynamic DRL scheduling algorithm is shown in **Figure 3.16**. As stated above, this scheduler depends on the size of the event window. The basis for the behavior of the proposed DRL multicontext scheduling algorithm is the use of the array *DRLArrayUtilization*, which represents the expected state (active DE classes or contexts)

```
DynamicDRLSchedulingAlgorithm (EW)
{
 ObtainDRLArrayUtilization(EW);
 K = NumberOfAvailableDRL();

 if K = 0 then
  DRLCell = GetLatestRequiredClass();
  Class = GetFirstClassNotInDRLArray(EW);
  DRL_Behaviour(DRLCell, Class);
 else
   CE = GetCurrentEvent();
  for Class = CE to CE+EW loop
   if ActiveClass(Class) = FALSE then
      DRLCell = GetFirstAvailableDRL(Class);
      DRL_Behaviour(DRLCell, Class);
   end if;
  end loop;
 end if;
}
```

Figure 3.16: DRL dynamic scheduling algorithm

of the DRL array within the event window. This array is obtained from the current state of the DRL array and the event window, using the function *ObtainDRLArrayUtilization*. Afterwards the algorithm calculates the number of DRL cells that will not be used within the event window (variable *K*). These *K* DRL cells (if there is anyone) are available for a class (context) switch. So, this is the first condition that the algorithm checks. If there are not any DRL cells available for a class switch, the algorithm selects (to be replaced) the DRL cell which has an active DE class that will be required latest. The algorithm also selects a DE class to be placed as active. The first DE class in the event stream which is not active within the DRL array will be selected. Finally, it performs the class switch with function *DRL_Behavior()*. On the other hand, if there are *K* DRL cells available for a class switch, the algorithm enters into a loop that goes through the complete event window. If it finds a DE class (associated with an event) which is not active within the DRL array, the algorithm selects the first available DRL cell to be set as active.

The proposed HW/SW co-design methodology was applied to the software acceleration of telecom broadband network simulation. In [12], a complete system based on this methodology has been implemented that also uses dynamic power management features to conserve power. The experiments conducted using this methodology indicates its effectiveness as a highly general purpose and thorough approach for the implementation of dynamic multifunction embedded systems using DRL.

4. EVALUATION AND SUMMMARY OF THE PROPOSED APPROACHES

Each of the approaches presented in the previous section build upon the previous ones by further refining the co-synthesis process and taking advantage of the recent developments in the architecture

of DRL. The CORDS approach is the earliest co-synthesis targeting dynamic reconfiguration, but it limits only a single task to be assigned to each context. The CRUSADE and Genetic Algorithm approaches remove this limitation by assigning more than one task at the same time. The concepts of early partial reconfiguration and incremental reconfiguration attempts to minimize the reconfiguration latency even further. Then a fine grained partitioning approach was presented that aims at optimizing the loops in the design specification. The next approach takes take power consumption minimization into consideration. All these approaches use static scheduling. Finally the last approach is the most general purpose in terms of partitioning and scheduling and considers the issue of energy consumption as well.

5. CONCLUSION

From the current state of research in the area of hardware/software co-synthesis using dynamically reconfigurable logic, it can be concluded that the promise of reconfigurable logic in reducing system cost and improving performance has being delivered. The issue of power savings has also been addressed and the practicality of using DRL in embedded systems design has been demonstrated by working prototypes. The area is now maturing and entering into mainstream embedded systems design with the availability of low delay reconfigurable devices.

Bibliography

[1] Giovanni De Micheli, Rolf Ernst, Wayne Wolf: ***Readings in Hardware Software Co-Design***. Morgan Kaufmann Publishers, CA, 2001

[2] Wayne H. Wolf, "***Hardware-Software Co-Design of Embedded Systems***", Proceedings of the IEEE, Vol. 82, No. 7, July 1994, pp. 967-989.

[3] Robert P. Dick, Niraj K. Jha, "***CORDS: Hardware-Software Co-Synthesis of Reconfigurable Real-Time Distributed Embedded Systems***", Proceedings of the International Conference on Computer Aided Design (ICCAD), November 1998, pp 62- 68.

[4] Bharat P. Dave, "***CRUSADE: Hardware/Software Co-Synthesis of Dynamically Reconfigurable Heterogeneous Real-Time Distributed Embedded Systems***", Proceedings of the Design, Automation and Test in Europe Conference, March 1999, pp. 97-104.

[5] K. Ben Chehida, M. Auguin, "***HW/SW Partitioning Approach For Reconfigurable System Design***", Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, October 2002, pp. 247-251.

[6] Byungil Jeong, Sungjoo Yoo, Sunghyun Lee, Kiyoung Choi, "***Hardware-Software Co-Synthesis for Run-time Incrementally Reconfigurable FPGAs***", Proceedings of Asia and South Pacific Design Automation Conference, January 2000, pp. 169-174.

[7] Yanbing Li, Tim Callahan, Ervan Darnell, Randolph Harr, Uday Kurkure, Jon Stockwood, "***Hardware-Software Co-Design of Embedded Reconfigurable Architectures***", Proceedings of the 37th conference on Design automation, June 2000, pp. 507-512.

[8] John Ried. Hauser, "***Augmenting a Microprocessor with Reconfigurable Hardware***", PhD. Dissertation, Computer Science, University of California, Berkeley

[9] Daler N. Rakhmatov, Sarma B. K. Vrudhula, "***Hardware-Software Bi-partitioning for Dynamically Reconfigurable Systems***", Proceedings of the tenth International Symposium on Hardware/software Co-design, May 2002, pp. 145-150.

[10] L. Shang, R. P. Dick, and N. K. Jha, "***SLOPES: Hardware-Software Co-Synthesis of Low Power Real-Time Distributed Embedded Systems with Dynamically Reconfigurable FPGAs***", revised for IEEE Transactions on Computer-Aided Design.

[11] Robert P. Dick, Niraj K. Jha, "***MOGAC: A Multiobjective Genetic Algorithm for Hardware-Software Co-Synthesis of Distributed Embedded Systems***", IEEE Transactions on Computer Aided Design, vol. 17, October 1998, pp. 920-935.

[12] Juanjo Noguera, Rosa M. Badia, "***HW/SW Codesign Techniques for Dynamically Reconfigurable Architectures***", IEEE Transactions on VLSI Systems, vol. 10, No. 4, August 2002, pp.399-415.

[13] Juanjo Noguera, Rosa M. Badia, "***System-Level Power-Performance Trade-Offs in Task Scheduling for Dynamically Reconfigurable Architectures***", Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems, October 2003, pp. 73-83.