Physically-Aware HW-SW Partitioning for Reconfigurable Architectures with Partial Dynamic Reconfiguration †

Sudarshan Banerjee Elaheh Bozorgzadeh Nikil Dutt
Center for Embedded Computer Systems
University of California, Irvine, CA, USA
{banerjee,eli,dutt}@ics.uci.edu

ABSTRACT

Many reconfigurable architectures offer partial dynamic configurability, but current system-level tools cannot guarantee feasible implementations when exploiting this feature. We present a physically aware hardware-software (HW-SW) scheme for minimizing application execution time under HW resource constraints, where the HW is a reconfigurable architecture with partial dynamic reconfiguration capability. Such architectures impose strict placement constraints that lead to implementation infeasibility of even optimal scheduling formulations that ignore the nature of these constraints. We propose an exact and a heuristic formulation that simultaneously partition, schedule, and do linear placement of tasks on such architectures. With our exact formulation, we prove the critical nature of placement constraints. We demonstrate that our heuristic generates high-quality schedules by comparing the results with the exact formulation for small tests and a popular, but placement-uanaware scheduling heuristic for larger tests. With a case study, we demonstrate extension of our approach to handle heterogenous architectures with specialized resources distributed between general purpose programmable logic columns. The execution time of our heuristic is very reasonable- task graphs with hundreds of nodes are processed in a couple of minutes.

Categories and Subject Descriptors: B.6.3 [C.1.3]
General Terms: Algorithms
Keywords: HW-SW partitioning, partial dynamic reconfiguration, linear placement

1. INTRODUCTION

Dynamic reconfiguration, often referred to as RTR (run-time reconfiguration) provides the ability to change hardware configuration during application execution. This enables a larger percentage of the application to be accelerated in hardware, hence reducing overall application execution time [12]. Modern-day SRAM-based FPGAs are examples of such hardware devices. Additionally, some FPGAs such as the Virtex devices from Xilinx [17] allow modification of only a part of the configuration (partial RTR). This is a very powerful feature specially for single-context FPGAs, by enabling the possibility of overlapping computation with reconfiguration to reduce the significant reconfiguration time overhead. Multicontext devices such as Morphosys [5] incur a lower overhead by paying a very significant area penalty to simultaneously store multiple contexts. Our work focuses on single-context devices where the dynamic reconfiguration overhead is very significant.

In this work, we consider the problem of task level HW-SW partitioning for a resource-constrained system, where the HW device has partial RTR capability. In a traditional codesign flow, HW-SW partitioning optimizes the design latency– subsequently the physical design stage places the tasks scheduled to HW on the underlying device. However, our target system architecture imposes strict linear placement constraints. Under such constraints, even an optimal schedule generated without considering the exact physical location of the task [7], may be physically unrealizable because of placement infeasibility.

This work makes several contributions:

- We demonstrate that existing approaches that do not consider physical task layout can result in unrealizable (infeasible) designs.
- We outline an exact approach that incorporates physical layout.
- We present a KLFM heuristic incorporating detailed linear placement that generates good results on a large set of benchmarks.
- We show applicability of our work to heterogenous architectures.

A key benefit of considering placement and multiple task implementations is the ability to extend our approach to consider heterogeneity with relatively minor modifications. Heterogeneity is a key aspect of modern reconfigurable architectures like the Virtex-II that contain dedicated resource columns of multipliers, block memories distributed between general purpose programmable logic columns. Such dedicated resources often lead to more area-efficient implementations that operate at a higher frequency. In a detailed case study of mapping a jpeg encoder task graph under resource constraints, we explore the benefits and issues with dynamic task implementations using heterogenous resources on such architectures.

2. RELATED WORK

HW-SW partitioning is an extensively studied problem with a plethora of approaches, including many KLFM-based approaches (Kernighan-Lin/Fiduccia-Matheyes, [15], [14]) such as [8], [11]. However, existing work often does not consider the special challenges posed by dynamic reconfiguration– partial RTR imposes more physical constraints that need to be incorporated explicitly.

Recently there has been work on simultaneous scheduling and placement for partially reconfigurable devices [1], [4]. However, they ignore key issues in run-time reconfiguration such as prefetch to overcome latency, the resource contention due to single reconfiguration controller, etc. With these simplifications, the problem becomes closer to rectangle packing [13]. Another approach to reducing the significant reconfiguration overhead is reuse, where the work often considers all tasks to be of equal area and focuses on exploiting similarity between a given set of scheduled tasks [2]. In

†This work was partially supported by NSF Grants CCR-0203813 and CCR-0205712

DAC 2005, June 13–17, 2005, Anaheim, California, USA.

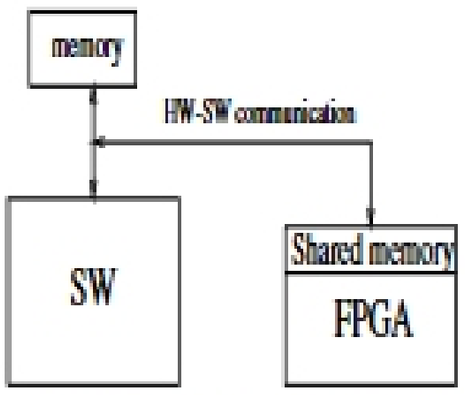

Figure 1: System architecture

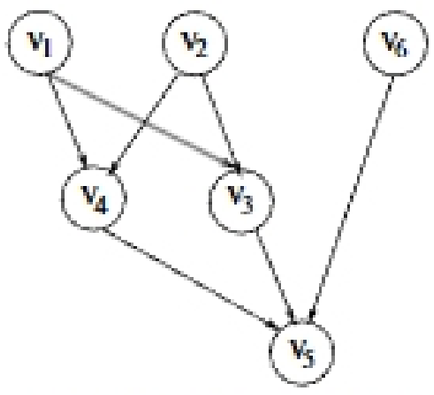

Figure 2: Dependency task graph

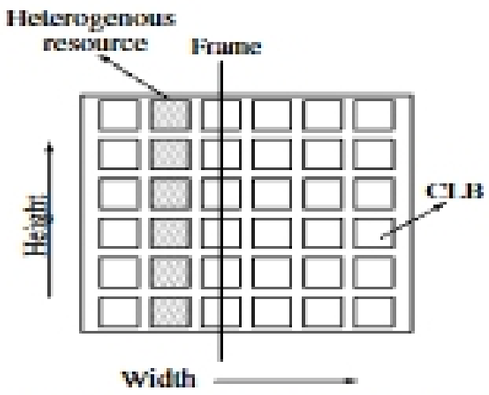

Figure 3: Heterogenous FPGA with partial RTR

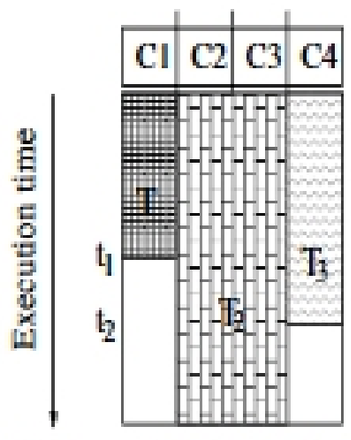

Figure 4: Simple infeasible

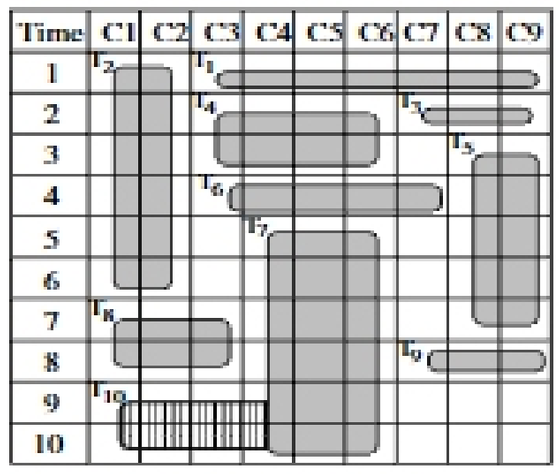

Figure 5: Detailed infeasible

our work, we currently do not exploit such resource-sharing across tasks. We focus on integrating key architectural constraints and placement considerations into the scheduling formulation for the more realistic scenario of varying task sizes.

Our work is most closely related to [6] and [7]. Mei et al. [6] present a genetic algorithm for partial RTR that considers columnar task placement. However, their approach does not consider prefetch or the single reconfiguration controller bottleneck. Jeong et al. [7] present an exact algorithm (ILP) and a KLFM-based approach. Their ILP considers prefetch and the single reconfiguration controller bottleneck– however, while scheduling, they do not consider the critical issue of physical task placement. We will demonstrate that an optimal formulation that does not simultaneously consider placement while scheduling can generate schedules which can not be placed and hence are not physically realizable. Another distinctive feature of our work compared to existing work is our consideration of heterogeneity in resources, a key feature of modern reconfigurable architectures.

3. PROBLEM DESCRIPTION

We consider the problem of HW-SW partitioning of an application specified as a task dependency graph extracted from a functional specification in a high-level language like C, VHDL, etc. In a task dependency graph (Figure 2), each vertex represents a task that can start execution only when all its ancestors have completed.

Our target system architecture as shown in Figure 1 consists of a SW processor and a dynamically reconfigurable FPGA with partial reconfiguration capability. The processor and the FPGA communicate by a system bus. We assume concurrent execution of the processor and the FPGA. We assume that the dynamically reconfigurable tasks on the FPGA communicate via a shared memory mechanism– this shared memory can be physically mapped to local on-chip memory and/or off-chip memory depending upon memory requirements of the application. Under this abstraction, communication time between two tasks mapped to the FPGA is independent of their physical placement. Thus, when adjacent tasks in the task graph are mapped to the same device (processor or FPGA), the communication overhead is considered insignificant, while tasks mapped to different devices incur a HW-SW communication delay.

On such a system architecture, a task can have multiple implementations: as a simple example, compiler optimizations like loop unrolling often result in a faster implementation with more HW area. Another example is the possibility of a very area-efficient implementation using dedicated resources like embedded memory.

Our objective is to minimize the execution time of the application while respecting the architectural and resource constraints imposed by the system architecture. Thus, our desired solution is a task schedule where each task is bound to HW or SW along with a suitable implementation point for each task.

Dynamically reconfigurable FPGA

Our target dynamically reconfigurable device as shown in Figure 3 consists of a set of configurable logic blocks (CLB) arranged in a two-dimensional matrix. Additionally, a limited number of specialized resource columns are distributed between CLB columns (the Xilinx Virtex-II architecture is an example of such a device). The basic unit of configuration for such a device is a frame spanning the height of the device. A column of resources consists of multiple frames. A task occupies a contiguous set of columns. Such a device is configured through a bit-serial configuration port like JTAG or a byte-parallel port. However, only one reconfiguration can be active at any time instant. The reconfiguration time of a task is directly proportional to the number of columns (frames) occupied by the task implementation.

4. KEY ISSUES IN SCHEDULING

4.1 Criticality of linear task placement

Each task implementation mapped to the target reconfigurable device occupies a set of adjacent columns. Under our abstraction that communication between such tasks is realized through a shared memory accessible from each task, task placement on such a device reduces to simple linear placement.

LEMMA 1. *For a given scheduled task graph with inter-task communication via shared memory and equal size tasks, a feasible and optimal placement is guaranteed and can be generated in polynomial time.*

PROOF. The problem for equal sized tasks reduces to graph coloring on interval graphs and thus efficient algorithms like left-edge algorithm can be applied [3]. More details in [16]. □

However, for tasks that occupy a different number of columns in the implementation, **placement feasibility is not guaranteed even with an exact algorithm**. (detailed explanation in [16]) In Figure 4 we demonstrate an instance of such infeasibility using an exact approach for partitioning and scheduling followed by linear placement for such multi-column tasks. This is a two-dimensional view of the task schedule where the Y-axis (length) corresponds to time, the X-axis (width) corresponds to number of columns. The FPGA has 4 columns and 3 tasks mapped onto it. Tasks T_1, T_2, T_3 occupy columns C_1, (C_2,C_3), and C_4 respectively. At time t_2, a model that does not consider placement information would indicate that 2 units of area were available. So a new task, say T_4, that requires 2 columns, could be scheduled at time t_2. However, this would be incorrect as 2 adjacent columns are not available at t_2.

In Figure 4, of course there is the opportunity for better placement by initially placing task T_2 into columns (C_3,C_4)– then, at

time t_2, 2 adjacent columns (C_1, C_2) would be available to place a 2 column task. However, the more detailed example in Figure 5 demonstrates that there are schedules that can not be placed by an optimal placement tool. At time step 9, task T_{10} needs 4 columns for execution- even though there are 6 columns available in the FPGA, 4 contiguous columns are not available. Note that changing the task placement at prior time-steps (for example swapping physical location of task T_3 with task T_4) would only lead to placement failure at a previous time-step. To achieve a feasible placement, the task schedule itself needs to change. Therefore, it is critical to integrate linear placement of the tasks into the scheduling formulation in order to generate feasible solutions.

4.2 Heterogeneous implementations

Modern FPGAs (such as the Xilinx Virtex-II) have heterogenous architectures containing columns of dedicated resources like embedded multipliers, embedded memory blocks. Usage of such specialized resources usually leads to more area-efficient and faster implementations. As an example, we consider post-routing timing data obtained from synthesizing a 2-dimensional DCT (discrete cosine transform) under columnar placement and routing constraints on the Virtex-II chip XC2V2000. While the heterogenous implementation with 3 CLB columns and 1 resource column has an operating frequency of 88 MHz, the homogenous implementation with 4 CLB columns is able to operate at only 64 MHz (we consider the adjacent column pair of BRAM (embedded memory) and MULTX18 (embedded multiplier) as a single resource column for generating numerical data).

However, these heterogenous resources are typically limited in number and present in specific locations. For instance, XC2V2000 has 48 CLB columns, but only 4 heterogenous resource columns. Since these resource columns are available only at fixed locations, they impose stricter placement constraints. Depending on where a task is placed, the HW execution time and area may vary significantly. This provides further motivation for considering linear placement as an integral aspect of HW-SW partitioning on reconfigurable architectures.

4.3 Scheduling for configuration prefetch

Configuration pre-fetch [9] is a powerful technique that attempts to overcome the significant reconfiguration penalty in single-context dynamically reconfigurable architectures by separating a task into reconfiguration and execution components. While the execution component is scheduled after data dependencies from parent tasks in the task graph are satisfied, the reconfiguration component is not constrained by such dependencies. This poses a significant challenge to any scheduling formulation that incorporates prefetch.

5. PROPOSED APPROACH

First, we modify the problem description to address the previous issues: We have a task graph with n tasks, where each task has multiple possible implementations. Each HW implementation of a task occupies a certain number of columns. We have one available SW processor, and a HW resource constraint of m HW columns for application mapping. Our objective is to find an optimal schedule where each task is bound to HW or SW, the task implementation is fixed, and, for HW tasks, the physical task location is determined.

ILP formulation: To understand the problem space and determine optimality, we first formulated an ILP (integer linear program) with key 0-1 variables $x_{i,j,k}$ denoting execution of task T_i starting at timestep j, leftmost column k, $r_{i,j,k}$ denoting reconfiguration of T_i, etc. The key constraints enforcing contiguity for multi-column tasks, configuration prefetch to reduce schedule length, resource constraints imposed by the single reconfiguration controller, etc. are explained in [16]. However, a commercial ILP solver (CPLEX) required an exorbitant amount of computation time to obtain an optimal solution even for relatively small problem instances. This motivated us to develop a heuristic approach that generates reasonably good-quality solutions with a computation effort many orders of magnitude lower– our heuristic generates quality solutions to problems with hundreds of tasks in a couple of minutes.

5.1 Heuristic formulation

Our approach is based on the well-known Kernighan-Lin/Fiduccia-Matheyes (KLFM) heuristic [15], [14] that iteratively improves solutions to "hard" problems by simple moves. At each step of the KLFM heuristic, the quality of a move needs to be evaluated. Similar to previous work in HW-SW partitioning such as [8], we evaluate the quality of a move by a scheduler. However, our target platform requires that our scheduler is specifically aware of the physical device architecture.

Code segment 1: **KLFM loop**

```
while (more unlocked tasks)
    for each unlocked task
        for each non-current implementation point
            calculate makespan by physically aware list-scheduling
    select & lock best (unlocked task, implementation point) tuple
    update best partition if new partition is better
```

Code segment 1 represents the KLFM kernel: while there are more unlocked tasks, the "best" task is chosen in every iteration of the loop. The kernel is itself repeatedly executed c times where c is a small constant, around 5-6. As can be seen above, our kernel considers multiple task implementation. In simple cases where each task has a single HW and a single SW implementation, a "move" in HW-SW partitioning usually implies moving the task to the other partition. In task implementations on FPGAs, multiple area-time tradeoff points are very common. Restricting a move to only *HW-SW*, or vice-versa would restrict the solution space. Thus we define a move as generic, possible between *any two implementation points* of a task, including HW-HW, HW-SW.

For the scheduler, we choose a simple list-scheduling algorithm. In a list-scheduler, at each stage there is a set of 'ready' nodes whose parents have been scheduled. The scheduler chooses the 'best' node based on some priority measure– the schedule quality depends strongly on priority assignment of nodes. Note that the scheduler is embedded inside the partitioner; thus, the scheduler always sees a bound graph where each task is assigned to HW or SW and hence the HW-SW communication on each edge is known.

We do simultaneous scheduling and placement– once a node is selected for scheduling, it is immediately placed onto the device. This ensures that all generated schedules are correct by construction. Thus, at every KLFM step, along with task binding, we also have the placed schedule available.

In traditional resource-constrained scheduling, priority functions like "nodes on critical path first" are applied uniformly to all nodes. But, on our target HW, factors that affect placement, such as configuration prefetch, play a key role in scheduling. So we propose that during task selection, processor tasks are compared between themselves on the simple basis of longest path, while FPGA tasks are compared using a more complex function. Key parameters of any such function are EST (earliest computation start time of task), EFT (earliest finish time), task area, and the longest path through the task, i.e, the function can be described as: f (EST, longest path, area, EFT). The EST computation embeds physical issues related to placement, resource bottleneck of single reconfiguration controller in the configuration prefetch process, etc., as described in more detail later.

Our observations indicate that it is usually more beneficial to first place tasks with narrower width (fewer columns): this leads