Integrating Physical Constraints in HW-SW Partitioning for Architectures With Partial Dynamic Reconfiguration

Sudarshan Banerjee, *Member, IEEE*, Elaheh Bozorgzadeh, *Member, IEEE*, and Nikil D. Dutt, *Senior Member, IEEE*

***Abstract*—Partial dynamic reconfiguration is a key feature of modern reconfigurable architectures such as the Xilinx Virtex series of devices. However, this capability imposes strict placement constraints such that even exact system-level partitioning (and scheduling) formulations are not guaranteed to be physically realizable due to placement infeasibility. We first present an exact approach for hardware–software (HW-SW) partitioning that guarantees correctness of implementation by considering placement implications as an integral aspect of HW-SW partitioning. Our exact approach is based on integer linear programming (ILP) and considers key issues such as configuration prefetch for minimizing schedule length on the target single-context device. Next, we present a physically aware HW-SW partitioning heuristic that simultaneously partitions, schedules, and does linear placement of tasks on such devices. With the exact formulation, we confirm the necessity of physically-aware HW-SW partitioning for the target architecture. We demonstrate that our heuristic generates high-quality schedules by comparing the results with the exact formulation for small tests and with a popular, but placement-uanaware scheduling heuristic for a large set of over a hundred tests. Our final set of experiments is a case study of JPEG encoding—we demonstrate that our focus on physical considerations along with our consideration of multiple task implementation points enables our approach to be easily extended 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 (partitioned, scheduled, and placed) in a couple of minutes.**

***Index Terms*—Hardware–software (HW-SW) partitioning, linear placement, partial dynamic reconfiguration.**

I. Introduction

DYNAMIC reconfiguration, often referred to as run-time reconfiguration (RTR) 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 [19]. Modern-day SRAM-based field-programmable gate arrays (FPGAs) are examples of such hardware devices. Additionally, some FPGAs such as the Virtex devices from Xilinx[1] 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 [11] 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 hardware–software (HW-SW) partitioning for a resource-constrained system, where the HW unit has partial RTR capability. Given an application represented as a task directed acyclic graph (DAG), our goal is to maximize application performance (minimize schedule length) when there exists a hard resource constraint on the amount of available configurable logic.

In a traditional codesign flow, HW-SW partitioning optimizes the design latency and is followed by the physical design stage that places the tasks scheduled to HW on the underlying device. However, for tasks mapped onto our target architecture, partial RTR capability imposes strict linear placement constraints. Under such constraints, an optimal schedule generated by a HW-SW partitioning approach that does not consider the exact physical location of the task during scheduling [13], may be physically unrealizable because of *placement infeasibility*.

Another key aspect of modern reconfigurable architectures like the Virtex-II is *heterogeneity*. Such architectures contain dedicated resource columns of multipliers, block memories, etc., distributed between general purpose programmable logic columns. Such dedicated resources often lead to more efficient implementations that operate at a higher frequency. It is important to consider the area-execution time tradeoffs arising from heterogeneity during HW-SW partitioning—for our problem, the placement restrictions due to heterogeneity pose an additional challenge.

- **Feasibility issue, exact approach**: With the previous two factors in mind, we first demonstrate that existing partitioning (and scheduling) approaches that do not consider physical task layout can result in unrealizable (infeasible) designs. This motivates us to present an exact approach to study the solution space. Our exact approach is an integer linear programming (ILP) formulation that incorporates physical layout into the HW-SW partitioning (and scheduling) problem. Our approach additionally integrates the key feature of configuration prefetch [16]—given the significant reconfiguration overhead of our target architecture, this feature is critical for minimizing schedule length.

Manuscript received October 10, 2005; revised April 3, 2006. This work was supported in part by the National Science Foundation under Grant CCR-0203813 and Grant CCR-0205712.

The authors are with the Center for Embedded Computer Systems, University of California, Irvine, CA 92697 USA (e-mail: banerjee@ics.uci.edu).

Digital Object Identifier 10.1109/TVLSI.2006.886411

[1][Online]. Available: www.xilinx.com

- **Heuristic approach**: While the ILP formulation is a key first step in exploring the problem space, the significant runtime makes it impractical for all but the simplest problems. So, we next present a Kernighan–Lin/Fiduccia–Matheyses (KLFM)-based heuristic that considers detailed linear placement as an integral part of scheduling. Our heuristic additionally considers the existence of *multiple task implementation points*, potentially arising from compiler optimizations. We compare our approach with the exact approach as well as with an approach that is insensitive to placement implications during scheduling—the experimental data over a large set of benchmarks (more than a hundred data points) confirms the necessity of considering placement implications as an integral part of scheduling on our target architecture. The runtime of our heuristic is very reasonable—task graphs with hundreds of nodes are *partitioned, scheduled, and placed* in a couple of minutes.
- **Heterogeneity**: A key benefit of considering placement and multiple task implementations is the ability to extend our approach to consider heterogeneity with relatively minor modifications. 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.

II. Related Work

HW-SW partitioning is an extensively studied problem with a plethora of approaches. This includes ILP-based exact approaches [20], genetic algorithm (GA)-based approaches [8], evolutionary algorithms (EA)-based approaches [3], and multiple KLFM-based ([24], [25]) approaches such as [14] and [18]. Of course, most of the existing work does not consider the special challenges posed by dynamic reconfiguration—the traditional HW-SW partitioning formulations implicitly assume that HW is *static*, i.e., the HW functionality cannot be modified during application execution. Partial RTR imposes additional placement constraints that need to be explicitly incorporated into the problem formulation.

Recently, approaches have been proposed for simultaneous scheduling and placement for partially reconfigurable devices [5], [10]. However, they do not consider key issues in runtime reconfiguration such as prefetch to overcome latency, the resource contention due to single reconfiguration controller, etc. In such work, the task reconfiguration is bundled along with task execution and treated as a single process—while such simplifications make the problem closer to rectangle packing [21], the proposed strategies are not applicable to *single-context* architectures with resource contention for reconfiguration and significant reconfiguration overheads.

There have been different proposals such as configuration compression, configuration caching [7], etc., to reduce the effect of large reconfiguration delays on such architectures. One of the popular approaches is configuration 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 [6]. In our paper, 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.

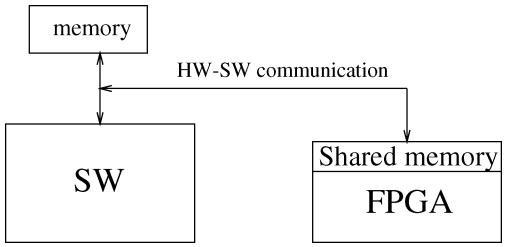


Fig. 1. System architecture.

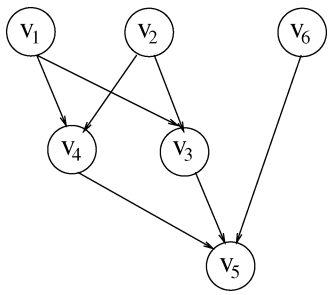


Fig. 2. Dependency task graph.

Our work is closely related to [12] and [13]. Mei *et al.* [12] 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.* [13] 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 during scheduling can generate schedules which can not be placed and, hence, are not physically realizable.

Last but not the least, a distinctive feature of our work compared to existing work is our consideration of heterogeneity in resources, a key feature of modern reconfigurable architectures.

III. Problem Description

We consider the problem of HW-SW partitioning of an application on the target system architecture shown in Fig. 1—the application is 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 (Fig. 2), each vertex represents a task. Each edge represents data that needs to be communicated from a parent task to a child task. Each task in the task graph can start execution only when all its immediate parents have completed and it has received all its input data from its parents.

A. Target System Architecture

Our target system architecture, as shown in Fig. 1, consists of two processing units: an SW processor and a dynamically reconfigurable FPGA with partial reconfiguration capability. The

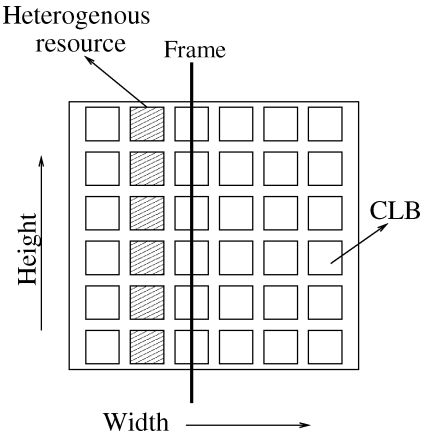


Fig. 3. Heterogenous FPGA with partial RTR.

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.[2] Under our abstraction, communication time between two tasks mapped to the FPGA is independent of their physical placement.

We also assume that all memory accesses by tasks executing on the processor are restricted to its local memory subsystem. When a task T_i executes on the processor, but one of its parent tasks T_j executes on the FPGA, the output data for T_j (input for T_i) needs to be transferred from the FPGA shared memory to the processor local memory incurring a communication overhead. Similarly, for a task executing on the FPGA, if one of its parent tasks executed on the processor, there is a communication delay for transferring data from the processor memory to the FPGA shared memory. Of course, for a task executing on the processor if all its parents executed on the processor, there is no penalty for data transfer. Thus, we need to consider HW-SW communication delay only for adjacent tasks mapped to different processing units.

B. Dynamically Reconfigurable FPGA

Our target dynamically reconfigurable HW unit as shown in Fig. 3 consists of a set of configurable logic blocks (CLB) arranged in a 2-D matrix. Additionally, a limited number of specialized resource columns are distributed between CLB columns. 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.

An example of such a dynamically reconfigurable HW unit is the Xilinx Virtex-II architecture. Reconfiguration delay for one column on a typical device (XC2V2000) is approximately 0.19 ms. In the Virtex-II architecture, there are dedicated columns of embedded multipliers (MULTX18), and block memories (BRAM) always placed adjacent to each other. In the rest of this paper, we consider the (MULTX18, BRAM) column pair as a single resource column for the purpose of generating sample numerical data on a representative architecture. Some of the Virtex devices (such as the Virtex-II Pro), have *hard* SW processors such as the PowerPC. However, all the Virtex devices are capable of instantiating the *soft* MicroBlaze processor.

C. Problem Parameters

On the target 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 area-efficient implementations using dedicated resources like embedded memory. Thus, each implementation point of a task can be summarized by the following set of parameters:

- execution time;
- area occupied in columns
  (for HW implementation points only);
- reconfiguration delay
  (for HW implementation points only);

and the device-related constraints can be summarized as follows:

- columnar implementations of dynamic tasks;
- single reconfiguration process;
- location of specialized resource columns
  (for heterogenous devices only).

D. HW-SW Partitioning Objective

Our objective for HW-SW partitioning 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 the HW unit or the SW processor, along with a suitable implementation point for each task.

Before presenting our proposed approach to solve this problem, in the next section, we take a detailed look at key issues such as implementation feasibility that are addressed by our proposed approach.

IV. Key Issues in Scheduling on Target Architecture

In this section, we present a detailed discussion on the key issues we have addressed in our formulation. First, we consider the criticality of considering physical constraints in a HW-SW partitioning formulation for a system with partial RTR.

A. Criticality of Linear Task Placement

In the target architecture, each *dynamic* task is implemented on a set of adjacent columns on the FPGA. Inter-task communication is realized through a shared memory accessible from

[2]In this work, we assume the shared memory mechanism provides sufficient bandwidth for a set of tasks concurrently executing on the FPGA. Also, detailed consideration of specialized inter-module communication structures such as those considered for the Erlangen Slot Machine [1] is beyond the scope of this work.

each task with the same latency and cost. Since this latency is identical for all the HW tasks and negligible compared to runtime reconfiguration overhead and HW-SW communication delay, inter-task communication delay for tasks mapped to the FPGA is not considered during HW-SW partitioning. This simplifies the placement of the tasks on the device to simple linear placement. Of course, since physical connectivity between tasks is not relevant under a shared memory abstraction, this linear placement problem is simpler compared to the linear placement problem in physical design where the objective is to minimize the total connectivity between the modules [27].

The linear task placement problem is formulated as follows:

- given a scheduled task graph under resource constraint and the size of the implementation for each task (in terms of the number of columns on a FPGA);
- find a *feasible* placement on reconfigurable hardware.

We look at this problem for two different cases. In the first case, we assume that each task occupies an identical number of columns. This assumption has been considered in previous work in dynamic reconfiguration such as [6]. In this case, feasible placement is guaranteed after tasks are scheduled on the FPGA under a total resource constraint.

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 is the same as track assignment on a set of intervals and graph coloring on interval graphs (which are perfect graphs) [9]. Each scheduled task represents an interval and each set of columns (equal to the size of tasks) represents a track. Since the graph is scheduled under a total number of columns, the number of resources available at each time is equal to the density of the tasks. Hence, by applying efficient algorithms for graph coloring on interval graphs (e.g., left-edge algorithm), a feasible placement can be found. □

Thus, task placement is trivial for tasks with identical size and can follow HW-SW partitioning. So, there is no need to integrate placement with HW-SW partitioning.

In the other case, we assume that tasks can occupy a different number of columns during implementation. After the tasks are scheduled, the feasibility of placement is not guaranteed even if it is checked with an exact algorithm. Similar to the first case, the placement problem is a track assignment problem for a set of intervals under the constraint that each interval gets assigned to a certain number of adjacent tracks. We can extend the aforementioned algorithm for track assignment based on a dynamic programming approach. While sweeping the time steps, we add the current interval to all existing feasible arrangements of already visited intervals. Due to adjacency constraint, some of those are not acceptable and the feasible assignments are pruned further. We continue until the end of the tracks. All the feasible combinations are examples of feasible placement. If no feasible combination is found, it implies that the current scheduled tasks do not have a feasible placement. The algorithm is linear in terms of the number of intervals but has a factorial growth on the number of tracks. The complexity of this problem is still an open problem. However, the exact solution can be obtained by the proposed extension to track assignment or using ILP solvers to check the feasibility of the placement. In this paper, our focus is on feasibility of placement after scheduling. We, thus, apply an exact solver to check the feasibility of the placement in order to show that the infeasibility in the placement comes from applying distinct consecutive stages of partitioning and placement rather than using suboptimal placement algorithms. Thus, for tasks that occupy a different number of columns in the implementation, *linear placement feasibility is not guaranteed even with an exact algorithm on a scheduled graph*.

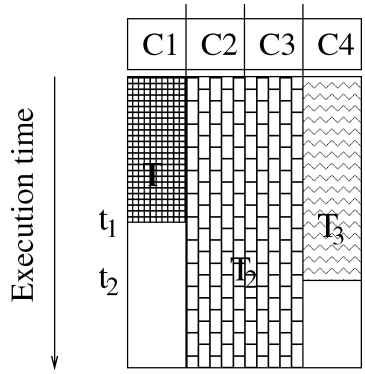


Fig. 4. Simple infeasible.

In Fig. 4, we demonstrate an instance of such infeasibility using an exact approach for partitioning and scheduling followed by linear placement for such multicolumn tasks. This is a 2-D view of the task schedule where the Y-axis (length) corresponds to time, the X-axis (width) corresponds to the number of columns. The FPGA has four columns and three tasks mapped onto it. Tasks T_1, T_2, and 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 two units of area were available. So a new task, say T_4, that requires two columns, could be scheduled at time t_2. However, this would be incorrect as two adjacent columns are not available at t_2.

In Fig. 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, two adjacent columns (C_1, C_2) would be available to place a two column task. However, the more detailed example in Fig. 5 demonstrates that there are schedules that cannot be placed by an optimal placement tool. At time step 9, task T_{10} needs four columns for execution—even though there are six columns available in the FPGA, four contiguous columns are not available. Note that changing the task placement at prior timesteps (for example, swapping physical location of task T_3 with task T_4) would only lead to placement failure at a previous timestep. 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.

B. Heterogeneity Considerations in Scheduling

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

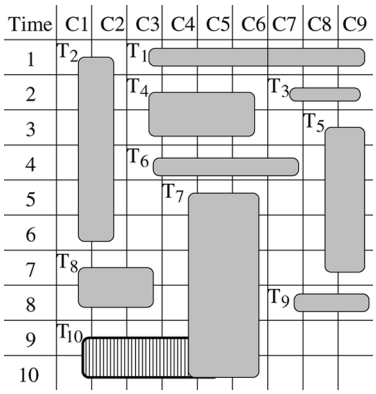


Fig. 5. Detailed infeasible.

post-routing timing data obtained from synthesizing a 2-D discrete cosine transform (DCT) under columnar placement and routing constraints on the Virtex-II chip XC2V2000. While the heterogenous implementation with three CLB columns and one resource column has an operating frequency of 88 MHz, the homogenous implementation with four 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 four 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.

C. Scheduling for Configuration Prefetch

Configuration prefetch [16] 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.

V. 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 an 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. In the rest of this section, we present an exact (ILP) formulation that solves this problem and follow up with a KLFM-based heuristic.

A. Notation

The problem input is a directed acyclic task dependency graph $G = (V, E)$. V is the set of graph vertices and E the set of edges. Each edge $e_{\rm ij}$ has one weight $ct_{\rm ij}$. ct_{ij} represents the HW-SW communication time, i.e., if v_i is mapped to SW and its child v_j is mapped to HW (or *vice versa*), $ct_{\rm ij}$ represents the time taken to transfer data between the SW and the HW unit. Each task T_i corresponding to vertex v_i has four weights (t_i^s, t_i^h, c_i, and $t_i^{\rm rf}$). t_i^s is the execution time of the task corresponding to v_i on the SW unit (processor). t_i^h, c_i, and t_i^{rf} are the execution time, area requirement in columns, and the reconfiguration overhead, respectively, for task T_i on the FPGA.

Our problem objective is to obtain an optimal mapping with minimal latency when the FPGA has at most $C_{\rm fpga}$ columns available for application execution.

B. ILP Formulation

In this section, we present an ILP that provides an exact solution to our problem. For ease of understanding, we restrict the ILP formulation to homogenous devices with single HW task implementation points only. As mentioned earlier, our work differs from existing ILPs in HW-SW partitioning, such as [20], in that we consider *linear* task placement as a key aspect—thus, our underlying model is essentially a 2-D grid where task placement is modeled along one axis while time is represented on the other axis. While this model is similar to existing ILP formulations for rectangular packing problems [23], issues such as configuration prefetch and the reconfiguration controller are unique to our problem and have not been considered in previous work on rectangular packing.

1) ILP Variables: We introduce the following set of 0–1 (decision) variables.

- $x_{i,j,k} = 1$, if task T_i starts execution on FPGA at time-step j, and k is leftmost column occupied by T_i;
  $= 0$, otherwise.
- $y_{i,j} = 1$, if T_i starts execution on processor in time-step j;
  $= 0$, otherwise.
- $r_{i,j,k} = 1$, if reconfiguration for task T_i starts at time-step j, and k is leftmost column occupied by T_i;
  $= 0$, otherwise.
- $in_{i_1,i_2} = 1$, if tasks T_{i_1} and T_{i_2} are mapped to different computing units and, thus, incur a HW-SW communication delay;
  $= 0$, otherwise.

Some of the constraints necessitate the introduction of additional binary variables to represent logical conditions. All such variables are represented as b.

The ranges of the variable indices are of course determined by the problem input, i.e.,

$$i \in (1 \ldots \text{number of tasks})$$
$$j \in (1 \ldots \text{upper bound on schedule length})$$
$$k \in (1 \ldots \text{number of FPGA columns}).$$

2) Constraints:

1) **Uniqueness constraint:**
Each task can start (is executed) exactly once

$$\forall i, \quad \sum_j \left(y_{i,j} + \sum_k (x_{i,j,k}) \right) = 1. \tag{1}$$

2) **Processor resource constraint:**
Processor executes, at most, one task at a time

$$\forall j, \quad \sum_i \sum_{m=j-t_i^s+1}^{j} (y_{i,m}) \leq 1. \tag{2}$$

3) **Partial dynamic reconfiguration constraints:**
   a) Every task needs, at most, one reconfiguration; and reconfiguration is not needed if task i executes on processor[3]

$$\forall i, \quad \sum_j \left(y_{i,j} + \sum_k (r_{i,j,k}) \right) \leq 1. \tag{3}$$

   b) **Resource constraints on FPGA:** total number of columns being used for task executions and number of columns being reconfigured is limited by the total number of FPGA columns

$$\forall j, \quad \sum_i \sum_k \left(\sum_{m=j-t_i^h+1}^{j} \sum_{n=k-c_i+1}^{k} (x_{i,m,k}) + \sum_{m=j-t_i^{rf}+1}^{j} \sum_{n=k-c_i+1}^{k} (r_{i,m,k}) \right) \leq C_{\text{fpga}}. \tag{4}$$

Note that in this constraint we do not explicitly consider the effect of *configuration prefetch*, where there are columns that have been reconfigured, but not yet executed. Subsequent constraints e) and f) ensure correctness for columns used in configuration prefetch.

   c) At every timestep j, at most single task is being reconfigured

$$\forall j, \quad \sum_i \sum_{m=j-t_i^{rf}+1}^{j} \sum_k (r_{i,m,k})) \leq 1. \tag{5}$$

Note that in this equation we do not need to consider the number of columns required for this task.

   d) At every timestep j, mutual exclusion of execution and reconfiguration for every column

$$\forall j, \forall k, \quad \sum_i \left(\sum_{m=j-t_i^h+1}^{j} \sum_{n=k-c_i+1}^{k} (x_{i,m,n}) + \sum_{m=j-t_i^{rf}+1}^{j} \sum_{n=k-c_i+1}^{k} (r_{i,m,n}) \right) \leq 1. \tag{6}$$

Note that this is a key step that enforces **contiguity**. The inner term $\sum_{n=k-c_i+1}^{k}(r_{i,m,n})$ ensures that if a task T_i requires c_i columns for reconfiguration (execution), it can proceed only when a contiguous set of c_i columns are available.

   e) If reconfiguration is needed for task T_i, execution of task T_i must start in the same column. Additionally, execution can start only after the reconfiguration delay

$$\forall i, \forall k, \quad \sum_j (r_{i,j,k}) = 1 \Longrightarrow \sum_j (j * r_{i,j,k}) + t_i^{rf} <= \sum_j (j * x_{i,j,k}). \tag{7}$$

We can rewrite the previous constraint as the following set of constraints:

$$f(X) = \sum_j (r_{i,j,k}) > 0$$

$$g(X) = \sum_j (j * x_{i,j,k} - j * r_{i,j,k}) - t_i^{rf} \geq 0$$

$$\text{if } (f(X) > 0) \text{ then } g(X) \geq 0.$$

This enables us to apply the *if–then* transformation as in [26].[4]

Note that in this equation and (3), we do not include the reconfiguration time for the initial set of tasks placed on the device. This enables us to accurately compare results with a traditional HW-SW partitioning formulation where execution time does not include system setup time of reconfiguration for the set of tasks placed on the device.

   f) For every column, there must be a reconfiguration between two task executions using this column. This key clause is necessary to account for the potential (gap) between reconfiguration and execution caused by *configuration prefetch*, discussed in the previous section.

We solve this problem by computing the difference between the execution start times and reconfiguration start times for all tasks that have used this column till a particular timestep. If a task execution is using this column at this timestep, this difference must necessarily be less than the execution start time of this task. For potentially better visual understanding, we focus on a specific column and consider the start times (reconfiguration and execution) of all tasks that use this column to be arranged in a linear sequence. The reconfiguration start times are assigned negative sign,

[3]Additional clarification for this constraint is included in the explanation of (7).

[4]If–then transform for the constraint if $(f(X) > 0)$, then $g(X) \geq 0$

$$-g(X) \leq Mb$$
$$f(X) \leq M(1-b)$$
$$b \in (0,1)$$

where M is a large number such that $f(X) \leq M$, $-g(X) \leq M$ for X satisfying all other constraints.

and the execution start times are assigned positive sign. The numbers in the sequence are obviously increasing in magnitude, but of differing signs. The constraints e) and f) together ensure that positive and negative numbers alternate, i.e., a task is reconfigured before execution. Additionally, they ensure that a column reconfigured but not yet executed does not get overwritten by another reconfiguration

$$\forall k, \forall j, \quad \sum_i \sum_{m=j-t_i^h+1}^{j} \sum_{n=k-c_i+1}^{k} (x_{i,m,n}) = 1 \Longrightarrow \sum_i \sum_{n=k-c_i+1}^{k} \sum_{m=1}^{j} (m * x_{i,m,n} - m * r_{i,m,n}) \leq \sum_i \sum_{n=k-c_i+1}^{k} \sum_{m=j-t_i^h+1}^{j} (m * x_{i,m,n}). \quad (8)$$

Similar to (7), we can rewrite (8), and apply the if–then transform.

g) *Simple placement constraint:* a task can start execution only if there are sufficient available columns to the right

$$\forall i, \forall j, \forall k \in (C_{\text{fpga}} - c_i + 1 \ldots C_{\text{fpga}}), \quad x_{i,j,k} = r_{i,j,k} = 0. \quad (9)$$

4) **Interface (communication) constraints:**
For each directed edge e_{i_1,i_2}, communication (interface) overhead is incurred if tasks T_{i_1} and T_{i_2} are mapped to different computing units, i.e., one is mapped to the processor and the other is mapped to the FPGA.
If task T_{i_1} is mapped to the processor, $\sum_j (y_{i_1,j}) = 1$. Thus, the communication overhead corresponding to the edge e_{i_1,i_2} is incurred under the following set of conditions:

$$\text{Either,} \quad \left(\sum_j (y_{i_1,j}) = 1 \text{ and } \sum_j (y_{i_2,j}) = 0 \right)$$

$$\text{Or,} \quad \left(\sum_j (y_{i_1,j}) = 0 \text{ and } \sum_j (y_{i_2,j}) = 1 \right).$$

That is, if we introduce a new variable

$$P_{i_1,i_2} = \sum_j (y_{i_1,j}) + \sum_j (y_{i_2,j}) + in_{i_1,i_2}.$$

P_{i_1,i_2} can only belong to the set $\{0, 2\}$.
Thus, the communication constraint is simply

$$\forall \text{edges}(i_1, i_2), \quad P_{i_1,i_2} = 2 * b \quad (10)$$

where b is a binary 0–1 variable.

5) **Precedence constraints:**
For each directed edge e_{i_1,i_2}, the start time for task T_{i_2} is necessarily at least the sum of the start time of task T_{i_1}, the execution time of task T_{i_1}, and the HW-SW communication time if any, i.e., $\forall \text{edges}(i_1, i_2)$

$$\sum_j \left(\left(\sum_k (j * x_{i_1,j,k}) \right) + j * y_{i_1,j} \right) + \sum_j \left(\sum_k (t_{i_1}^h * x_{i_1,j,k}) + t_{i_1}^s * y_{i_1,j} \right) + ct_{i_1,i_2} * in_{i_1,i_2} \leq \sum_j \left(\sum_k (j * x_{i_2,j,k}) + j * y_{i_2,j} \right). \quad (11)$$

6) **Objective function to minimize schedule length:**
This is equivalent to minimizing the start time of the sink task T_n

$$\text{minimize} \sum_j \left(j * y_{n,j} + \sum_k (j * x_{n,j,k}) \right).$$

Of course, by introducing simple additional constraints that force the sink task T_n to execute on the processor and all tasks to have 0 communication delay with the sink task, the objective function can be simply written as

$$\text{minimize} \sum_j (j * y_{n,j}).$$

Along with the necessary constraints, we also introduce **additional constraints** that help significantly in reducing the time the ILP solver needs to find a solution.

7) **Tighter placement constraints:**
For every column k, at every time instant j, total number of executions using this column so far is at most 1 less than the total number of reconfigurations

$$\forall k, \forall j, \quad \sum_{n=k-c_i+1}^{k} \sum_{m=1}^{j} \sum_i (r_{i,m,n} - x_{i,m,n}) \leq 1. \quad (12)$$

8) **Tighter timing constraints:**
ASAP, ALAP constraints.

3) Extending the ILP for Multiple, Heterogenous Implementations: While our ILP formulation is based on single homogenous task implementations, we believe that it can be easily extended for single heterogenous task implementations by a simple preprocessing step that adds extra placement constraints to the homogenous formulation. Extensions for handling multiple task implementation points is more challenging. One crude but effective way would be to represent each $x_{i,j,k}$ as a linear sum of a set of 0–1 variables representing the different possible task implementations. Then all product terms of the form $c = a * b$ obtained by substituting the $x_{i,j,k}$ terms in the homogenous implementation can be linearized by using Fortet's linearization method [22].

C. Heuristic Approach

While our ILP formulation enabled us to study the problem space, its implementation using a commercial ILP solver

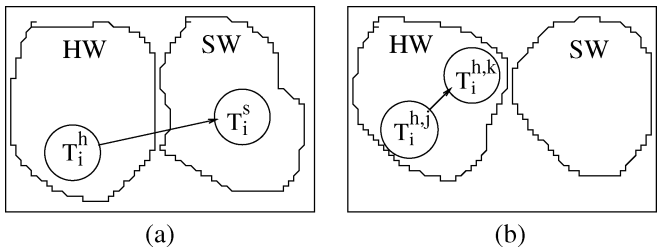


Fig. 6. Moves in HW-SW partitioning with multiple implementation points.

(CPLEX) required a very significant 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. We obtain quality solutions to problems with hundreds of tasks in a couple of minutes with our heuristic.

1) Heuristic Outline: Our approach is based on the well-known KLFM heuristic [24], [25] 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 [14], we evaluate the quality of a move by a scheduler. However, our target platform requires that our scheduler is aware of the physical and architectural constraints of the underlying device.

Code Segment 1: KLFM loop

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

In **Code Segment 1**, we present our adaptation of the KLFM kernel. Essentially this is the outer loop of the heuristic: 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 in **Code Segment 1**, our kernel considers multiple task implementation points. In simple cases where each task has a single HW and a single SW implementation, a "move" in HW-SW partitioning 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. In Fig. 6(a) we see an example of a traditional HW-SW partitioning move, where a move consists of selecting the SW implementation T_i^s of the task instead of selecting the HW implementation of the task T_i^h. However, in Fig. 6(b), we see a move that consists of selecting an alternate HW implementation point $T_i^{h,k}$ instead of $T_i^{h,j}$ because this leads to the most improvement in the objective function.

For the scheduler, we choose a simple list-scheduling algorithm as shown in **Code Segment 2**. 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.

Code segment 2: Choose best schedulable task

```
For each schedulable task,
    compute (EST), earliest start time of computation
            (EFT), earliest finish time of computation
Choose task that maximizes f(EST, longest path, area, EFT)
```

In traditional resource-constrained scheduling, priority functions like "nodes on critical path first" are applied uniformly to all nodes. But, given the special characteristics of our target HW, it is undesirable to use the same priority assignment function uniformly for nodes. 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 earliest computation start time of task (EST), earliest finish time (EFT), task area, and the longest path through the task, i.e., the function can be described as

$$f(\text{EST, longest path, area, EFT}).$$

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 to the possibility of being able to accomodate more tasks without needing dynamic reconfiguration. Similar considerations for other key parameters lead us to define f as a linear priority assignment function:

$$f = -A * \text{columns} - B * \text{EST} + C * \text{pathlength} - D * \text{EFT}.$$

Note that components for which it is preferable to have smaller magnitude, such as earlier start time (EST) or fewer columns, have a negative weightage while pathlength has a

Task	HW time	SW time	HW area
1	5	23	3
2	2	9	3
3	2	11	2
4	3	14	1
5	2	10	2
6	3	7	4

Fig. 7. Task parameters.

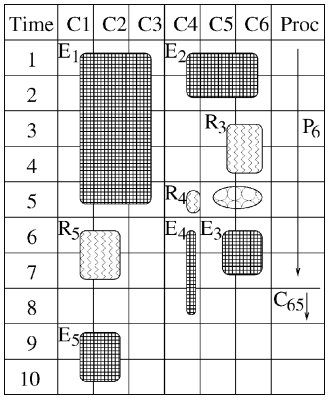


Fig. 8. Optimally placed.

positive weightage. Pathlength is of course the classical "critical path" priority function that is often used as the single node selection criterion in list scheduling.

2) Placement and EST Computation: To illustrate the effectiveness as well as the challenge posed by configuration prefetch to placement and scheduling, consider the task graph shown in Fig. 2, and its associated parameters in Fig. 7. The HW area is specified as the number of homogenous (CLB) columns. The execution times (for HW and SW), and the reconfiguration delay are suitably normalized such that each timestep corresponds to the reconfiguration delay for one column. Thus, the (normalized) reconfiguration delay of a task is identical to the HW area of the task. Additionally, for this example, we assume that any HW-SW communication incurs one unit of (normalized) delay.

Under a resource constraint of six homogenous columns, the optimal solution to our problem of minimizing latency is given by the task schedule and physical task location as shown in Fig. 8. In this schedule, each execution (and reconfiguration if needed) component of a task is represented as a rectangle of fixed size, such that the length is the execution (or reconfiguration) time of the task implementation while the width is the number of columns required.

In Fig. 8, E_i and R_i represent the execution start time and reconfiguration start time, respectively, for vertex v_i. C_{ij} represents HW-SW communication between task v_i and v_j. P_i represents execution of task v_i on the processor. For this example, with static HW-SW partitioning, the schedule length would be 26 with vertices v_1, v_3, and v_4 mapped to HW and the remaining vertices mapped to SW. Since partial dynamic reconfiguration capability with prefetch improves the schedule length to 10, prefetch is a key consideration.

However, a key challenge is posed by the gap between R_3 and E_3 illustrating the idle time interval of columns $C5$ and $C6$ required for an optimal schedule: in this interval the FPGA column has been reconfigured, but the task cannot start execution as its dependencies have not been satisfied yet. Note that the earliest E_3 can start is at time step 6. So, if we forced R_3 to start at time step 4 and contiguous to E_3, then either R_4 would need to be separated from E_4 or the schedule length would increase.

This idle time interval is part of scheduling in that we would prefer to have a schedule with minimum idle time where resources are underutilized. Since the extent of the interval can not be determined *a priori*, placement is complicated: if we consider the aggregate $(\text{time} \times \text{area})$ rectangle occupied by a task in the 2-D view, where the aggregate rectangle consists of both the execution and reconfiguration component of a task, this is a rectangle of unknown length. Thus, with prefetch, we are unable to directly apply rectangular packing algorithms from work like [21].

Another key issue in EST computation is the resource bottleneck of a single reconfiguration controller. The reconfiguration for a task can start only when enough area is available *and* the reconfiguration controller is free. The goal is to complete reconfiguration before task dependencies are satisfied, leading to minimization of schedule length. However, realistically, it is not possible to hide the overhead for all tasks that need reconfiguration—in such cases, task execution is scheduled as soon as its reconfiguration ends.

In **Code Segment 3**, we present our approach to EST computation that addresses the issues we previously discussed.

Code Segment 3: Compute EST for task bound to FPGA

```
find earliest time slot where task can be placed
reconfig start = earliest time instant space and reconfig
     controller are simultaneously available.
if ((reconfig start + reconfig time) < dependency time)
     // reconfiguration latency hidden completely: possibility
     // of timing gap between reconfig end and execution start
   EST = earliest time parent dependencies satisfied
else     // not possible to completely hide latency
   EST = end of reconfiguration
```

Our goal is to find the earliest time slot when the task can be scheduled, subject to the various constraints. We proceed by first searching for the earliest instant when we can have a feasible task placement, i.e., enough adjacent columns are available for the task. Once we have obtained a feasible placement, we proceed to satisfy the other constraints. If the reconfiguration controller was available at the instant the space becomes available, then the reconfiguration component of the task can proceed immediately. Otherwise, the reconfiguration component of the task has to wait till the reconfiguration controller becomes free.

TABLE I
BASIS FOR NUMERICAL DATA

HW unit	similar to XC2V2000, organized as a CLB matrix of 56 rows and 48 columns
SW unit	PowerPC processor operating at 400 MHz
Communication bus	64-bit wide PLB operating at 133 MHz
Frames/CLB column	22 frames (total 1456 frames on the entire device)
Reconfiguration time	17.01 ms for full device (SelectMAP port@50 MHz)
Reconfig frequency	66 MHz (maximum suggested)
Reconfig delay/column	22/1456 * 17.01 * 50/66 = 0.19 ms

Once the reconfiguration component is scheduled, we check to see if the execution component can be immediately scheduled subject to dependency constraints. As an example, we consider EST computation of task T_3 in Fig. 8 when tasks T_1 and T_2 have been scheduled and placed. The initial search shows a feasible placement starting at time 3 and the reconfiguration controller is free, so reconfiguration for T_3 can start immediately and finishes at time 4. However, the execution component can be scheduled only at time 6 when its dependency is satisfied. In this case, EST computation indicates that it is possible to completely hide the reconfiguration overhead for the task.

The EST computation, thus, embeds the placement issues and resource constraints related to reconfiguration. As discussed earlier, the scheduler assigns task priorities based on this information, leading to high-quality schedules, as shown in our experimental section.

3) Comments on Current Implementation: The first search for the earliest feasible time instant is currently implemented as a simple sweep through all active time instants (when an event has been scheduled).[5] At each time instant, we represent the resource constraint as a simple array with each array entry in one of two states—free or used. (Note that the number of active time instants is $O(n)$, where n is the number of tasks.) Thus, the search for space to fit a task is equivalent to bin-packing, where we choose the *best* bin (array location) to place an item (a task). We implemented various bin-packing algorithms such as first-fit, best-fit, etc. Our initial set of experiments indicated that first-fit worked well, so all our results in the experimental section are based on first-fit packing. A subsequent detailed set of experiments (also presented in the experimental section) confirmed that the difference between first-fit and best-fit was negligible. However, best-fit needs significantly more expensive computation during the space-search confirming that our choice of first-fit is reasonable.

4) Heterogeneity: One key benefit of considering linear placement and multiple task implementations in our heuristic is the ease with which we were able to extend our approach to consider scheduling onto heterogenous devices.

To adapt our approach for heterogeneity, the primary change required is in the search for space to fit a task. We achieve this by simply adding a type descriptor for each column in our resource description. Thus, all resource queries at a time instant check the type descriptor of a column while looking for available space at that instant. Since the key implication of a heterogenous resource is to constrain placement, we did some simple initial preprocessing to make our searches more efficient.

5) Worst Case Complexity: Consideration of placement as an integral part of HW-SW partitioning guarantees correctness of implementation. However, it does increase the worst case complexity of HW-SW partitioning.

For an area constraint of C columns, our current simplistic implementation of the EST computation has a worstcase complexity (for a single task) of $O(n^2C)$, where n is the number of tasks. In the list scheduler, at each step, the *best* task is chosen based on this computation, i.e., $O(n)$ such EST computations are required to select the next task to be scheduled. Since the list scheduler has "n" such steps, the worstcase complexity of each list scheduler invocation is $O(n^4C)$. For the simple case of one HW and one SW implementation of a task, the list scheduler is called $O(n^2)$ times in the main KLFM loop shown in Code Segment 1. Thus, the overall worstcase complexity is $O(n^6C)$. While this seems to be a polynomial of a significantly high degree, execution time measurements presented in our experimental section indicate a runtime of a couple of minutes for our largest experiments on graphs with hundreds of nodes.

VI. EXPERIMENTS

We conducted a wide range of experiments to demonstrate the validity of our formulation and the schedule quality generated by our heuristic. We also conducted a detailed case study of the JPEG encoding algorithm, where we explored heterogeneity in the context of multiple task implementation points. Note that we are concerned with statically determining the best runtime schedule for a HW-SW system under resource constraints, where the HW has partial dynamic reconfiguration capability. Thus, while it is possible for the example to fit all our JPEG tasks in a suitably sized device, for our experimental purposes, we assume a resource constraint less than the aggregate HW size of all tasks leading to the necessity of HW-SW partitioning.

A. Experimental Setup

The following assumptions in Table I form the basis of our numerical data.

Area and timing data for key tasks like DCT and IDCT, was obtained by synthesizing tasks under columnar placement and routing constraints on the XC2V2000, similar to the methodology suggested for "reconfigurable modules." Software task execution time on the PowerPC processor is typically 3–5 times slower than the HW implementation of the task. HW-SW com-

[5]An event refers to start (end) of task execution/reconfiguration.

TABLE II
FEASIBILITY RESULTS AND HEURISTIC QUALITY FOR SMALL TESTS

	Placement-Unaware		Placement-Aware	
Testcase	T_{opt}^{area}	Feas.	T_{opt}	T_{heu}
tg1	10	Y	10	11
tg5	25	**NO**	26	26
Mean-value	21	Y	21	21
tg7	20	Y	20	20
tg10	27	**NO**	28	29
FFT	25	Y	25	25
tg11	36	**NO**	38	41
tg12	14	**NO**	15	18
4-band eq	27	Y	27	27

munication time was estimated by simply dividing the aggregate amount of data transfer by the bus speed. As an example, data transfer time for a 256×256 block of 8-bit pixels in a typical image processing application is estimated as

$$256 * 256 * 8/64 \text{ cycles at } 133 \text{ MHz} = 0.06 \text{ ms}.$$

Note that HW-SW communication time for even this significant volume of data transfer is only around 30% of the reconfiguration overhead for a single CLB column; thus, for generating synthetic experiments, we assumed that HW-SW communication time was quite low compared to task reconfiguration time.

B. Experiments on Feasibility

Table II shows experimental results on feasibility for a set of synthetic task-graphs and well-known graph structures like fast Fourier transform (FFT), meanval, etc. These test cases were reasonably small graphs with between 10–15 vertices such that we could generate optimal results with the ILP. For each test, we assumed that the number of columns available for task mapping was approximately 20%–30% of the aggregate area of all tasks mapped to hardware. For these tests, one unit of time is the reconfiguration time for a single column.

In Table II, T_{opt} denotes the schedule length obtained with our ILP formulation, $T_{\text{opt}}^{\text{area}}$ denotes the schedule length obtained from an exact formulation that considers available HW area instead of exact task placement (i.e., placement unaware) [13]. As Table II shows, in some cases, $T_{\text{opt}}^{\text{area}}$ is shorter than T_{opt}, but **in these cases, the schedules were physically unrealizable** with exact placement, while our ILP (T_{opt}) guarantees placement through correct by construction.

C. Experiments on Heuristic Quality

For each of the initial set of experiments, we also generated results with our proposed heuristic, as denoted by T_{heu} in Table II. The data indicates that for the small cases, T_{heu} corresponds to schedules that are reasonably close in quality to the exact solution.

For analysis of schedule quality generated by our heuristic on larger test cases, we generated a set of problem instances with suitable modifications to task graphs for free (TGFF) [17]. In these tests, each task had a single homogenous implementation point. In subsequent discussions, $v20$, $v80$, etc., denote sets of

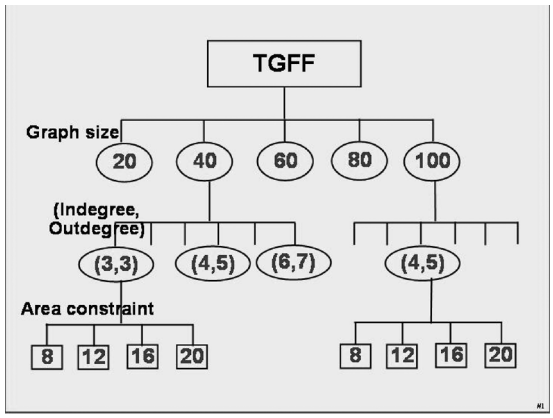


Fig. 9. Synthetic experiments.

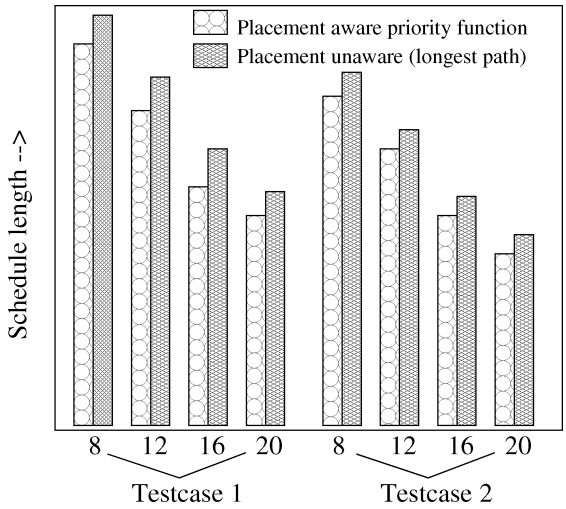


Fig. 10. Sample experiments for $v60$.

graphs that have approximately 20 nodes, 80 nodes, etc. These sets were generated by varying the graph parameters such as indegree, outdegree. For each individual test case belonging to a set like $v20$, we varied the area constraint from 8 to 20 columns in steps of 4 to generate a problem instance. The resulting space of over a hundred experiments is shown in Fig. 9.

For each generated problem instance, we compared the schedule length generated by our placement-aware heuristic with that generated by the placement unaware "longest path first" (LPF) heuristic. The LPF heuristic is widely used in resource-constrained scheduling to assign higher priorities to tasks on critical paths. Note that LPF is used only for priority assignment at each scheduling step—once a task is selected, the same linear placement approach ensures correct schedules and hides the reconfiguration latency, if possible.

In Fig. 10, we present a sample of the tests we conducted. For two test graphs in set $v60$, we show schedule length data corresponding to a total of eight problem instances. To present the aggregate data for the complete set of experiments, we define $T_{\text{longest\_path}}$ as the schedule length generated by LPF for a problem instance, and the quality criterion indicating improvement (decrease) in schedule length for each problem instance

TABLE III
AGGREGATE IMPROVEMENTS IN SCHEDULE LENGTH

Test group	Few cols (8,12)	More Cols (16,20)	Avg gain
v20	6.07%	6.79%	6.43%
v40	5.44%	10.64%	8.04%
v60	10.36%	10.56%	10.46%
v80	11.68%	13.64%	12.66%
v100	16.68%	19.09%	17.89%
Avg gain	10.05%	12.15%	11.09%

TABLE IV
COMPARISON OF FIRST-FIT VERSUS BEST-FIT

Test group	Few cols (8,12)	More Cols (16,20)	Avg gain
v20	0.0%	0.0%	0.0%
v40	0.0%	-0.25%	-0.12%
v60	0.14%	-0.02%	0.06%
v80	-0.26%	-0.07%	-0.17%
v100	0.26%	0.55%	0.40%
Avg gain			0.03%

when our placement-aware priority function is used compared to placement-unaware LPF as

$$\text{Gain} = 100 * (T_{\text{longest\_path}} - T_{\text{heu}})/T_{\text{heu}}.$$

Fig. 10 shows that our placement-aware priority function *consistently* generates better schedules. Table III summarizes the result for 120 problem instances. Each entry in the table represents data from a set of instances. As an example, the entry corresponding to the row labelled $v60$ and column labelled "More Cols (16,20)" is 10.56%. This implies that for a set of problem instances where the graph size is approximately 60 nodes and the resource constraint was set at 16 and 20 columns, the average improvement in schedule length generated by our heuristic over LPF was 10.56%.

As is clear from Table III, while a simple longest path heuristic works reasonably well with small graphs and few columns, our heuristic clearly generates superior (shorter) schedules, both with increasing problem size. The key difference is that LPF also tries to improve schedule length by prefetch, but only after selecting the task to be scheduled, while our heuristic considers placement implications in task selection.

1) First-Fit Versus Best-Fit: Similar to our previous table, we compare the quality difference between first-fit placement and best-fit placement by the measure

$$\text{Gain} = 100 * (T_{\text{best}} - T_{\text{first}})/T_{\text{first}}.$$

Table IV indicates that the quality difference between using a first-fit placement policy and a best-fit placement policy is negligible. However, the best-fit placement incurs additional computational overhead in the EST computation. This confirms our choice of the first-fit placement policy as suitable.

2) Runtime of Heuristic: Table V shows the average runtime of our approach (in seconds) for the experiments with an area-constraint of 20 columns. The measurements were done on a 502-MHz Sparcv9 processor (SunOS 5.8). While the runtime of our placement-aware approach grows with increase in area constraint, we believe that the data, corresponding to our largest experiments, is a fair representation of the expected runtime in reasonable scenarios.

TABLE V
RUNTIME OF PROPOSED APPROACH

Test group	Average run-time(s) 20 columns
v20	0.2
v40	2.0
v60	22
v80	90
v100	180

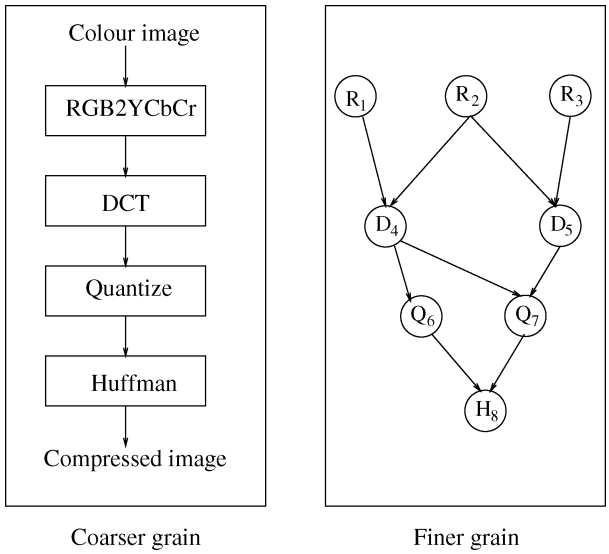


Fig. 11. Task graph for JPEG encoder.

D. Case Study of JPEG Encoder

We, next, conducted a detailed analysis for the JPEG encoding algorithm shown in Fig. 11 under resource constraints. We obtained data for tasks like quantize and Huffman, by synthesizing the tasks under placement and routing constraints. For each task, we obtained implementation points with only homogenous resources and with heterogenous resources. We assumed that the SW implementation for each task was approximately four times slower than the HW implementation using only homogenous resources. With only homogenous implementations, the total area occupied by the tasks in the coarse grain task graph in Fig. 11 was 11 columns. We assumed a resource constraint of 8 columns was available for mapping the task set.

Numerical data on the significant reconfiguration time for a CLB column confirms observations from previous researchers [4] that execution time for a task operating on a $8 * 8$ block of 8-bit data is orders of magnitude lower than the reconfiguration overhead of such tasks. So, all our schedule length data is for processing a larger block corresponding to a 256×256 color image.

Table VI presents a summary of schedule length estimates (in milliseconds) we generated from various experiments. The first row (16.74 ms) represents our initial experiment of HW-SW partitioning of the coarse-grain graph—in this experiment the HW does **not** have dynamic reconfiguration capability. The next

TABLE VI
SCHEDULE LENGTH FOR DIFFERENT HW-SW PARTITIONING OF JPEG ENCODER

	Experiment	Latency (ms)
Coarse-grain graph	HW-SW partitioning (no RTR)	16.74
	HW-SW + partial RTR	9.9
	HW-SW + partial RTR + perfect prefetch	9.04
Fine-grain graph	HW-SW + partial RTR (single homogenous implementation point)	7.51
	Multiple implementation points	6.82
	Best implementation points	9.58

row (9.9 ms) represents the experiment where we consider the HW to have partial RTR capability. It clearly demonstrates the potential for performance improvement with partial RTR. For this experiment, we assumed that there was no configuration prefetch, i.e., reconfiguration for a task was done exactly before its execution. In the third experiment (9.04 ms), where we add configuration prefetch to partial RTR capability, there is additional performance improvement.

We subsequently exposed more parallelism by making multiple copies of tasks like DCT based on our knowledge that data blocks can be independently processed by such tasks. The remaining results from the fourth row onwards corresponds to experimental data for the finer-grain task graph. The fourth row (7.51 ms) represents the results generated by our heuristic on the finer-grain graph—this is optimal for this representation.

1) Experiment on Heterogeneity: For the next experiment in the fifth row (6.82 ms), we considered that the resource constraint of eight columns now included one specialized resource (heterogenous) column, i.e., the new resource constraint was a set of seven CLB columns and one resource column. Each task was allowed to have either a homogenous implementation or a heterogenous implementation.

In the schedule generated by our heuristic, some of the tasks are bound to their faster heterogenous implementations while others are bound to slower homogenous implementations. This experiment demonstrates the exploration capability of our heuristic in considering multiple task implementations while mapping onto a heterogenous device with partial dynamic reconfiguration.

One important observation from our experiment with heterogeneity was that the relative location of the specialized resource column strongly affects the schedule length. Specifically for our first-fit placement policy, we observed that specialized resource columns located near the left edge of the device (where the first fit algorithm initially tries to place tasks) lead to inferior schedule lengths.

2) Best Implementation Points Only: For the final experiment in row 6 (9.58 ms), we restricted tasks to only their best implementation points. Since the best implementation points are often heterogenous, the schedule length showed significant degradation because of contention for the dedicated resources.

Overall, our case study confirms the importance of considering physical and architectural (heterogenous) constraints in an HW-SW partitioning algorithm for a partially reconfigurable device. It additionally confirms that partitioning (and scheduling) algorithms targeted towards such devices need to have the capability of selecting between multiple task implementations, some of which might be using specialized resources.

VII. CONCLUSION

In this paper, we focussed on physical and architectural constraints imposed on dynamically reconfigurable architectures by partial reconfiguration feature. We first formulated an exact approach based on ILP. With the help of this exact approach, we demonstrated that ignoring linear task placement constraints imposed by partial dynamic reconfiguration can result in schedules that are optimal but physically unrealizable. Unlike existing ILP-based approaches to HW-SW partitioning, our formulation simultaneously places tasks while scheduling—it also considers the key feature of configuration prefetch for maximizing performance along with the resource contention due to a single reconfiguration mechanism.

Next, we proposed a placement-aware HW-SW partitioning heuristic based on the well-known KLFM paradigm for partitioning. Our proposed heuristic simultaneously partitions, schedules, and does linear placement of tasks on the target device. As a key step of partitioning, our approach selects among multiple task implementation points. A wide range of synthetic experiments and a detailed case study of JPEG encoding validates the quality of solutions generated by our proposed heuristic.

Placement and consideration of multiple implementations in partitioning make it easy to extend our approach to heterogeneity, a key feature in modern FPGAs. The case study on JPEG encoding demonstrates the capability of our approach in selecting between heterogenous and homogenous task implementations while mapping a given application onto a heterogenous device. Finally, the runtime of our approach is reasonable: task graphs with hundreds of nodes are processed (partitioned, scheduled, placed) in a couple of minutes.

Our approach has powerful capabilities, but there is scope for improvement in our current implementation in both solution quality and in the theoretic algorithmic complexity by investigating sophisticated placement techniques and data structures. Next, our approach currently assumes that sufficient bandwidth is available for concurrently executing tasks—we realize this may not hold in all possible situations and adding bandwidth considerations would make our work more complete. Finally, our heuristic currently is focused on homogenous implementations. In the future, we will focus on issues leading to high-quality solutions in heterogenous scenarios.

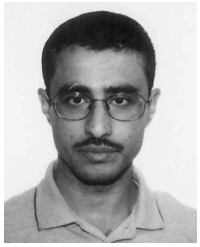
ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their helpful comments that significantly improved the clarity of their paper. A special thanks goes to the reviewer who carefully went through every single equation of the ILP and pointed out major typographical errors that would have made some key equations incorrect.

REFERENCES

[1] C Bobda, M. Majer, A. Ahmadiniya, T. Haller, A. Linarth, and J. Teich, "The Erlangen slot machine: Increasing flexibility in FPGA-based reconfigurable platforms," in *Proc. Field Program. Technol.*, 2005, pp. 37–42.

[2] S. Banerjee, E. Bozorgzadeh, and N. Dutt, "Physically-aware HW-SW Partitioning for reconfigurable architectures with partial dynamic reconfiguration," in *Proc. Des. Autom. Conf.*, 2005, pp. 335–340.
[3] C. Haubelt, S. Otto, C. Grabbe, and J. Teich, "A system-level approach to hardware reconfigurable systems," in *Proc. Asia South Pacific Des. Autom. Conf.*, 2005, pp. 298–301.
[4] J. Noguera and R. M. Badia, "Power-performance trade-offs for reconfigurable computing," in *Proc. IEEE/ACM/IFIP Int. Conf. Hardw.-Softw. Codes. Syst. Synt.*, 2004, pp. 116–121.
[5] P.-H. Yuh, C.-L. Yang, Y.-W. Chang, and H.-L. Chen, "Temporal floorplanning using the T-tree formulation," in *Proc. Int. Conf. Comput.-Aided Des.*, 2004, pp. 300–305.
[6] S. Ghiasi and M. Sarrafzadeh, "Optimal reconfiguration sequence management," in *Proc. Asia South Pacific Des. Autom. Conf.*, 2003, pp. 359–365.
[7] Z. Li, "Configuration management techniques for reconfigurable computing," Ph.D. dissertation, Dept. Elect. Comput. Eng., Northwestern University, Evanston, IL, 2002.
[8] K. B. Chehida and M. Auguin, "HW/SW partitioning approach for reconfigurable system design," in *Proc. Int. Conf. Compilers Arch. Synth. Embedded Syst.*, 2002, pp. 247–251.
[9] J. L. Ramirez-Alfonsin and B. A. Reed, Eds., *Perfect Graphs*. New York: Wiley, 2001.
[10] S. P. Fekete, E. Kohler, and J. Teich, "Optimal FPGA module placement with temporal precedence constraints," in *Proc. Des. Autom. Test Eur.*, 2001, pp. 658–667.
[11] H. Singh, G. Lu, E. M. C. Filho, R. Maestre, M.-H. Lee, F. J. Kurdahi, and N. Bagherzadeh, "MorphoSys: Case study of a reconfigurable computing system targeting multimedia applications," in *Proc. Des. Autom. Conf.*, 2000, pp. 573–578.
[12] B. Mei, P. Schaumont, and S. Vernalde, "A hardware-software partitioning and scheduling algorithm for dynamically reconfigurable embedded systems," in *Proc. ProRisc Workshop CKTS, Syst. Signal Process.*, 2000 [Online]. Available: http://www.imec.be/reconfigurable/pdf/prorisc_00_hardware.pdf
[13] B. Jeong, S. Yoo, S. Lee, and K. Choi, "Hardware-software cosynthesis for run-time incrementally reconfigurable FPGAs," in *Proc. Asia South Pacific Des. Autom. Conf.*, 2000, pp. 169–174.
[14] K. S. Chatha and R. Vemuri, "An iterative algorithm for hardware-software partitioning, hardware design space exploration, and scheduling," *J. Des. Autom. Embedded Syst.*, vol. V-5, pp. 281–293, 2000.
[15] M. Kaul and R. Vemuri, "Optimal temporal partitioning and synthesis for reconfigurable architectures," in *Proc. Des. Autom. Test Eur.*, 1998, pp. 389–396.
[16] S. Hauck, "Configuration pre-fetch for single context reconfigurable processors," in *Proc. ACM/SIGDA Int. Symp. Field Programm. Gate Arrays*, 1998, pp. 65–74.
[17] R. P. Dick, D. L. Rhodes, and W. Wolf, "TGFF: Task graphs for free," in *Proc. Int. Workshop Hardw.-Softw. Codes.*, 1998, pp. 97–101.
[18] F. Vahid and T. D. Le, "Extending the Kernighan-Lin heuristic for hardware and software functional partitioning," *J. Des. Autom. Embedded Syst.*, vol. V-2, pp. 237–261, 1997.
[19] M. J. Wirthlin, "Improving functional density through run-time circuit reconfiguration," Ph.D. dissertation, Electr. Comput. Eng. Dept., Brigham Young Univ., Laie, HI, 1997.
[20] R. Niemann and P. Marwedel, "An algorithm for hardware/software partitioning using mixed integer linear programming," *J. Des. Autom. Embedded Syst.*, pp. 165–193, 1997.
[21] H. Murata, K. Fujiyoshi, S. Nakatake, and Y. Kajitani, "Rectangle-packing based module placement," in *Proc. Int. Conf. Comput.-Aided Des.*, 1995, pp. 472–479.
[22] P. Hansen, B. Jaumard, and V. Mathon, "Constrained non-linear 0-1 programming," *ORSA J. Comput.*, vol. 5, no. 2, pp. 97–119, 1993.
[23] J. E. Beasley, "An exact two-dimensional non-guillotine cutting tree search procedure," *Op. Research*, vol. V-33, pp. 49–64, 1985.
[24] C. M. Fiduccia and R. M. Mattheyses, "A linear-time heuristic for improving network partitions," in *Proc. Des. Autom. Conf.*, 1982, pp. 175–181.
[25] B. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *Bell Syst. Techn. J.*, vol. V-29, pp. 291–307, 1970.
[26] W. L. Winston and M. Venkataraman, *Introduction to Mathematical Programming*, 4th ed. Boston, MA: Thomson Brooks Cole Publishers, 2003.
[27] M. Sarrafzadeh and C. K. Wong, *An Introduction to VLSI Physical Design*. New York: McGraw-Hill, 1994.



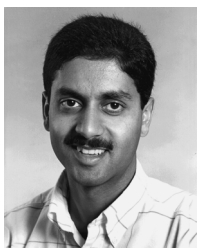
Sudarshan Banerjee (M'99) received the B.Tech. degree in computer science and engineering from the Indian Institute of Technology, Kharagpur, India, and the M.Tech. degree in computer science and engineering from the Indian Institute of Technology, Kanpur, India, in 1992 and 1995, respectively. He is currently pursuing the Ph.D. degree at University of California, Irvine.

His doctoral studies follows an extensive career in Research and Development for industry-leading logic verification tools (as an employee of Synopsys, Mountain View, CA, and Cadence, Noida, India). His current areas of interest include partitioning and scheduling for HW-SW codesign, and dynamically reconfigurable field-programmable gate arrays (FPGAs).

Elaheh Bozorgzadeh (S'99-M'03) received the B.S. degree in electrical engineering from the Sharif University of Technology, Tehran, Iran, in 1998, the M.S. degree in computer engineering from Northwestern University, Evanston, IL, in 2000, and the Ph.D. degree in computer science from the University of California, Los Angeles, in 2003.

She is currently an Assistant Professor in the Department of Computer Science, University of California, Irvine. Her research interests include VLSI computer-aided design (CAD), design automation for embedded systems, and reconfigurable computing.

Prof. Bozorgzadeh is a member of the ACM.



Nikil D. Dutt (SM'96) received the Ph.D. degree in computer science from the University of Illinois at Urbana-Champaign, Urbana, in 1989.

He is currently a Professor of Computer Science and Electrical Engineering and Computer Science at the University of California, Irvine (UCI) and is affiliated with the following centers at UCI: CECS, CPCC, and CAL-IT2. His research interests include embedded systems design automation, computer architecture, optimizing compilers, system specification techniques, and distributed embedded systems. He currently serves as Editor-in-Chief of *ACM Transactions on Design Automation of Electronic Systems* (TODAES) and as Associate Editor of *ACM Transactions on Embedded Computer Systems* (TECS).

Dr. Dutt was an ACM SIGDA Distinguished Lecturer during 2001–2002, and an IEEE Computer Society Distinguished Visitor for 2003–2005. He has served on the steering, organizing, and program committees of several premier Computer-Aided Design (CAD) and Embedded System Design conferences and workshops, including ASPDAC, CASES, CODES+ISSS, DATE, ICCAD, ISLPED, and LCTES. He serves or has served on the advisory boards of ACM SIGBED and ACM SIGDA, and is Vice-Chair of IFIP WG 10.5. He is a recipient of the Best Paper Awards at CHDL89, CHDL91, VLSIDesign 2003, CODES+ISSS 2003, CNCC 2006, and ASPDAC-2006.