HW-SW partitioning for architectures with partial dynamic reconfiguration

Sudarshan Banerjee Elaheh Bozorgzadeh Nikil Dutt
Center for Embedded Computer Systems
University of California, Irvine, CA, USA
Irvine, CA 92697-3425,USA
{banerjee,eli,dutt}@ics.uci.edu

CECS Technical Report #05-#02

April, 2005

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 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 ILP (integer linear programming) 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 enable our approach to easily 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.

Contents

List of Figures

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 [16]. Modern-day SRAM-based FPGAs are examples of such hardware devices. Additionally, some FPGAs such as the Virtex devices from Xilinx [25] 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 [8] 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 unit has partial RTR capability. Given an application represented as a task DAG (directed acyclic graph), our goal is to maximize application performance (minimize schedule length) when there exists a 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 while scheduling [10], 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 trade-offs 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 above 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 ILP (integer linear programming) formulation that incorporates physical layout into the HW-SW partitioning (and scheduling) problem. Our approach additionally integrates the key feature of configuration prefetch [13] – given the significant reconfiguration overhead of our target architecture, this feature is critical for minimizing schedule length.

Heuristic approach: While the ILP formulation is a key first step in exploring the problem space, the significant run-time makes it impractical for all but the simplest problems. So, we next present a KLFM-based heuristic (Kernighan-Lin/Fiduccia-Matheyes) 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 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 run-time of our heuristic is very reasonable – task graphs with hundreds of nodes are *partitioned, scheduled, 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.

2 Related work

HW-SW partitioning is an extensively studied problem with a plethora of approaches. This includes ILP (Integer Linear Programming)-based exact approaches [17], GA (genetic algorithm) based approaches [5], and, multiple KLFM-based approaches (Kernighan-Lin/Fiduccia-Matheyes [22], [21]) such as [11], [15]. 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 can not be modified during application execution. Partial RTR imposes additional placement constraints that need to be explicitly incorporated into the problem formulation.

Recently there has been work on simultaneous scheduling and placement for partially reconfigurable devices [2], [7]. However, they do not consider key issues in run-time 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 makes the problem closer to rectangle packing [18], 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 [4], 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 [3]. In 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 [9] and [10]. Mei et al. [9] 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. [10] 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

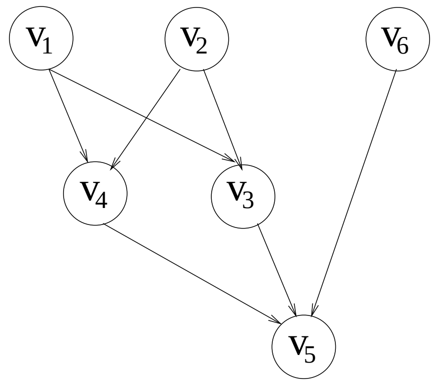

Figure 1. Dependency task graph

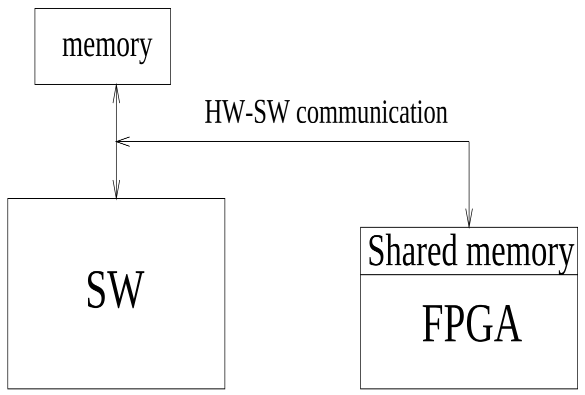

Figure 2. System architecture

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.

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 1), 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.

Target system architecture

Our target system architecture as shown in Figure 2 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 processing unit (processor or FPGA), the communication overhead is considered insignificant, while tasks mapped to different units incur a HW-SW communication delay.

Dynamically reconfigurable FPGA

Our target dynamically reconfigurable HW unit 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 basic unit of configuration for such a device is a frame spanning the height of the device. A column of resources

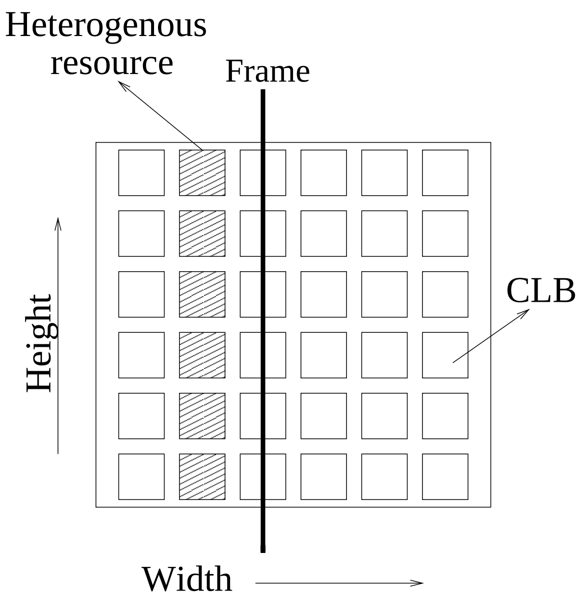

Figure 3. Heterogenous FPGA with partial RTR

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 architecture. In this architecture, there are dedicated columns of embedded multipliers (MULTX18), and block memories (BRAM) always placed adjacent to each other. In the rest of this report, 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.

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 a 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:

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

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.

4 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.

4.1 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 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 [24].

The linear task placement problem is formulated as:
• 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 [3]. In this case, feasible placement is guaranteed after tasks are scheduled on the FPGA under a total resource constraints.

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 same as track assignment on a set of intervals and graph coloring on interval graphs (which are perfect graphs) [6]. 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 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. $\square$

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 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 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**.

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

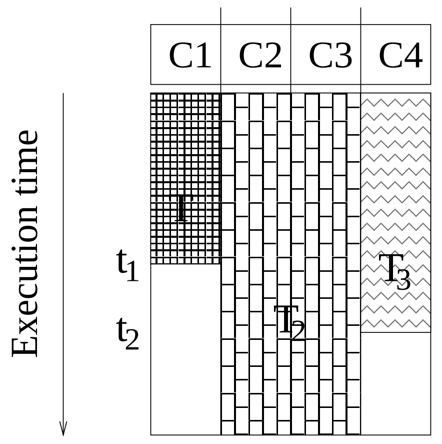

Figure 4. Simple infeasible

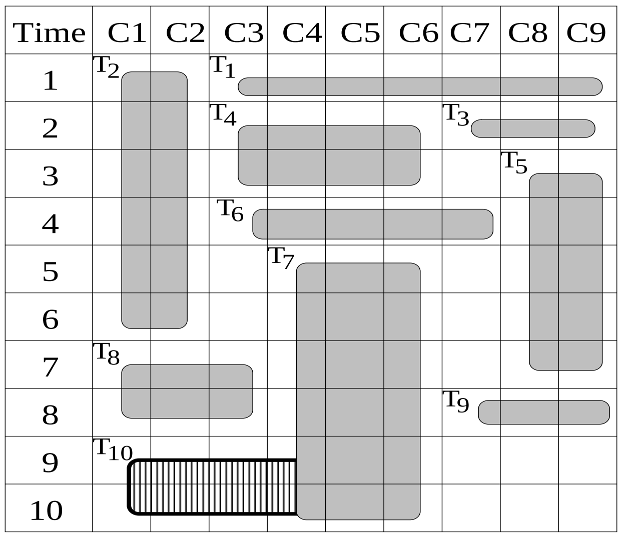

Figure 5. Detailed infeasible

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

5.1 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_{ij} has 1 weight ct_{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_{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 4 weights (t_i^s, t_i^h, c_i, t_i^{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, 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_{fpga} columns available for application execution.

5.2 ILP formulation

In this section, we present an ILP (integer linear program) 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 [17] in that we consider *linear* task placement as a key aspect – thus, our underlying model is essentially a two-dimensional grid where task placement is modelled along one axis while time is represented on the other axis. While this model is similar to existing ILP formulations for packing problems [20], issues such as configuration prefetch and the reconfiguration controller are unique to our problem and have not been considered in previous work on packing.

5.2.1 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 task 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 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 .. number of tasks)
$j \in$ (1 .. upper bound on schedule length)
$k \in$ (1 .. number of FPGA columns)

5.2.2 Constraints

1. Uniqueness constraint

Each task can start (is executed) exactly once.

$$\forall i, \quad \sum_j (y_{i,j} + \sum_k (x_{i,j,k})) = 1 \qquad (1)$$

2. Processor resource constraint

Processor executes at most one task at a time

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

3. Partial dynamic reconfiguration constraints

(a) Every task needs at most 1 reconfiguration; and, reconfiguration is not needed if task i executes on processor.

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

(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.

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

(c) At every time-step j , at most single task is being reconfigured.

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

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

(d) At every time-step j, mutual exclusion of execution and reconfiguration for every column.

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

Note that this is a critical 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 same column. Additionally, execution can start only after the reconfiguration delay.

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

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

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

This enables us to apply the *if-then* transformation as in [23]

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

where M is a large number such that $f(X) \leq M, -g(X) \leq M$ for X satisfying all other constraints. An appropriate value for M is $j_{max} * j_{max}$.

Note that in this equation and equation 3a, 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) When a task execution is using a column, the previous event on this column can never be another execution. Note that this possibility arises because of the gap (idle interval) possible between reconfiguration and execution, as discussed in the previous section.

We solve this problem by computing the difference between the reconfiguration start times and execution start times for all tasks that have used a column till a particular time-step. If this difference is more than the start time of the current executing task, then this column was previously used by another execution just prior to this execution, but, not reconfigured in between- this situation must never happen.

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

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

f(X) = $\sum_i \sum_{m=j-t_i^h+1}^{j} \sum_{n=k-c_i+1}^{k} (x_{i,m,n}) > 0,$
g(X) = $\sum_i \sum_{n=k-c_i+1}^{k} (\sum_{m=j-t_i^h+1}^{j} (m * x_{i,m,n}) +$
$\sum_{m=1}^{j} (m * r_{i,m,n} - m * x_{i,m,n}))) \geq 0$
if $(f(X) > 0)$ then $g(X) \geq 0$

and apply the *if-then* transformation as in the previous constraint.

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

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

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.

Either, $(\sum_j (y_{i_1,j}) = 1$ and $\sum_j (y_{i_1,j}) = 0$)
Or, $(\sum_j (y_{i_1,j}) = 0$ and $\sum_j (y_{i_1,j}) = 1)$.
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
$\forall edges(i_1, i_2)$,
$P_{i_1,i_2} = 2 * b \qquad (10)$

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} and the HW-SW communication time if any.

i.e, $\forall edges(i_1,i_2)$,

$$\sum_j((\sum_k(j*x_{i_1,j,k}))+j*y_{i_1,j})+ \sum_j(\sum_k(t^h_{i_1}*x_{i_1,j,k})+t^s_{i_1}*y_{i_1,j})+ct_{i_1,i_2}*in_{i_1,i_2} \leq \sum_j(\sum_k(j*x_{i_2,j,k})+j*y_{i_2,j}) \qquad (11)$$

6. Objective function to minimize schedule length

This is equivalent to minimizing the start time of the sink task T_n.

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

Of course, by introducing simple additional constraints that force the 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:

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 column k, at every time instant j,

total number of executions using this column so far is at most 1 less than total number of reconfigurations.

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

8. Tighter timing constraints

ASAP, ALAP constraints.

5.2.3 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 [19].

5.3 Heuristic approach

While our ILP formulation enabled us to study the problem space, its implementation using a commercial ILP solver (CPLEX) required an 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.

5.3.1 Heuristic formulation

Our approach is based on the well-known Kernighan-Lin/Fiduccia-Matheyes (KLFM) heuristic [22], [21] 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 [11], 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 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
```

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 above, 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 Figure 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 Figure 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 de-

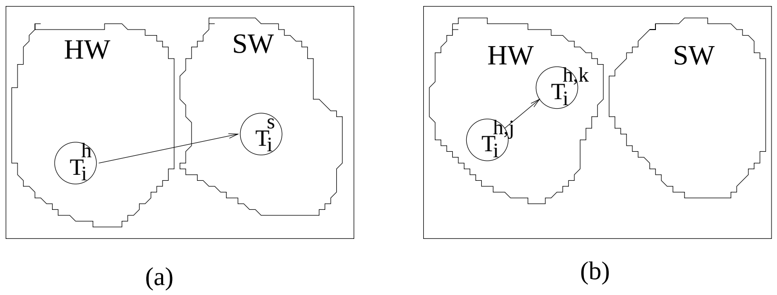

Figure 6. Moves in HW-SW partitioning with multiple implementation points

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

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

Figure 7. Task parameters

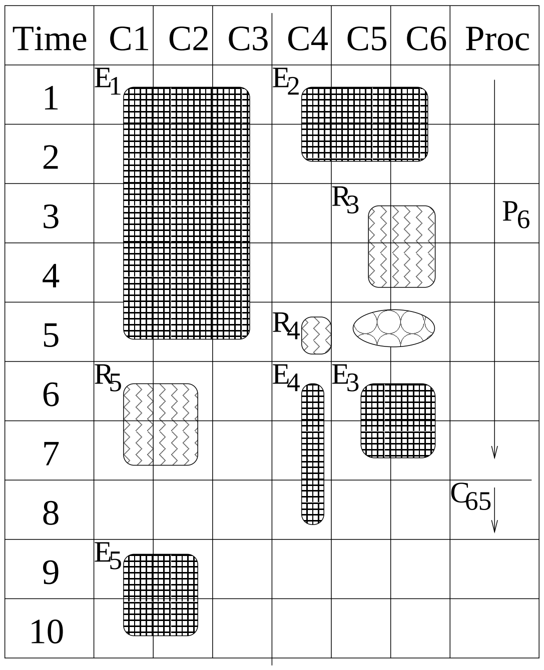

Figure 8. Optimally placed

$$-A * columns - B * EST + C * pathlength - D * EFT$$

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 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.

5.3.2 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 Figure 1, and its associated parameters in Figure 7. The HW area is specified as the number of homogenous (CLB) columns. For this example, we assume that any HW-SW communication incurs one unit of delay and the reconfiguration overhead of a task is equal to the HW area of the task.

Under a resource constraint of 6 homogenous columns, the optimal solution to our problem of minimizing latency is given by the task schedule and physical task location as shown in Figure 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 Figure 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 36 with vertices v_1 and v_2 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 $C5, C6$ required for an optimal schedule: in this interval the FPGA column has been reconfigured, but the task can not 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 resource are underutilized. Since the extent of the interval can not be determined apriori, placement is complicated: if we consider the aggregate $(time\ X\ area)$ rectangle occupied by a task in the two-dimensional 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 [18].

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 discussed above.

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. 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 Figure 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.

Comments on current implementation

The first search for earliest feasible time instant is currently implemented as a a simple sweep through all active time instants (when an event has been scheduled). 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 $O(n)$. To search for space to fit a task, we implemented various 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.

5.3.3 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.

5.3.4 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 worst-case complexity of $O(n^2C)$. Thus, the worst-case 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

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 (a total of 1456 frames on the entire device)
Reconfiguration time	17.01 ms for entire device (SelectMAP port at 50 Mhz);
Reconfiguration frequency	66 MHz (maximum suggested)
Reconfiguration overhead/CLB	22/1456 * 17.01 * 50/66 = 0.19 ms

Table 1. Basis for numerical data

list-scheduler is called $O(n^2)$ times in the main KLFM loop shown in Code Segment 1. Thus, the overall worst-case complexity is $O(n^6C)$. While this seems to be a polynomial of significantly high degree, execution time measurements presented in our experimental section indicate a run-time of a couple of minutes for our largest experiments on graphs wth hundreds of nodes.

6 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 run-time schedule for a HW-SW system under resource constraints, where the HW has partial dynamic reconfiguration capability. Thus, while it is possible for 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.

6.1 Experimental setup

The following assumptions in Table 1 form the basis of our numerical data:

Area and timing data for key tasks like DCT, 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 to 5 times slower than the HW implementation of the task. HW-SW communication time was estimated by simply dividing the aggregate amount of data transfer by the bus speed. As an example, data transfer time for a $256X256$ block of 8-bit pixels in a typical image processing application is estimated as:

256 * 256 * 8/64 cycles at 133 MHz = 0.06 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.

Testcase	Placement-Unaware		Placement-Aware	
	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

Table 2. Feasibility results and heuristic quality for small tests

6.2 Experiments on feasibility

Table 2 shows experimental results on feasibility for a set of synthetic task-graphs and well-known graph structures like 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 2, T_{opt} denotes the schedule length obtained with our ILP formulation, T_{opt}^{area} denotes the schedule length obtained from an exact formulation that considers available HW area instead of exact task placement (i.e, placement-unaware) [10]. As Table 2 shows, in some cases, T_{opt}^{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.

6.3 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 2. 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 TGFF [14]. In these tests, each task had a single homogenous implementation point. In subsequent discussions, *v*20, *v*80, etc, denote sets of 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 *v*20, 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 Figure 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

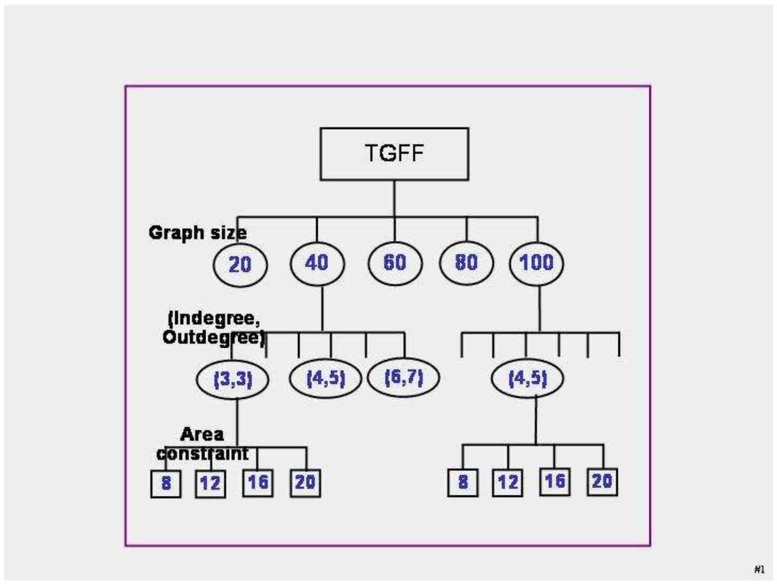

Figure 9. Synthetic experiments

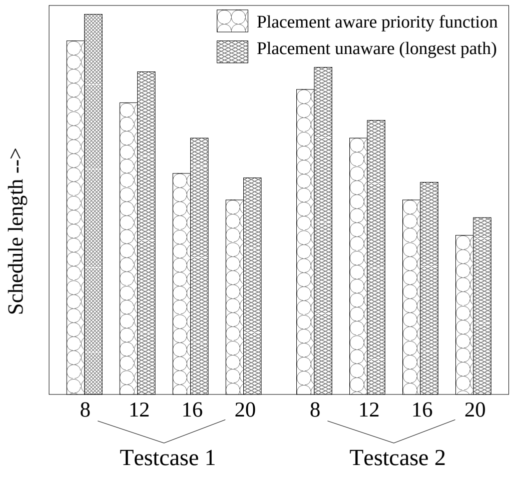

Figure 10. Sample experiments for v60

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 3. Aggregate improvements in schedule length

tasks on critical paths. Note: 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 Figure 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 8 problem instances. To present the aggregate data for the complete set of experiments, we define $T_{longest\_path}$ 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 when our placement-aware priority function is used compared to placement-unaware LPF as:

$$100 * (T_{longest\_path} - T_{heu})/T_{heu}$$

Figure 10 shows that our placement-aware priority function *consistently* generates better schedules. Table 3 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 "Avg gain (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 6.86%.

As is clear from Table 3, 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.

6.3.1 First-fit Vs best-fit

Similar to our previous table, we compare the quality difference between first-fit placement and best-fit placement by the measure:

$$100 * (T_{best} - T_{first})/T_{first}$$

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%

Table 4. Comparison of first-fit Vs best-fit

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

Table 5. Run-time of proposed approach

Table 4 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.

6.3.2 Run-time of heuristic

Table 5 shows the average run-time 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 run-time 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 run-time in reasonable scenarios.

6.4 Case study of JPEG encoder

We next conducted a detailed analysis for the JPEG encoding algorithm Figure 11 under resource constraints. We obtained data for tasks like quantize, 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 4 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 Figure 11 was 11 columns. We assumed a resource constraint of 8 columns was available for mapping the task set.

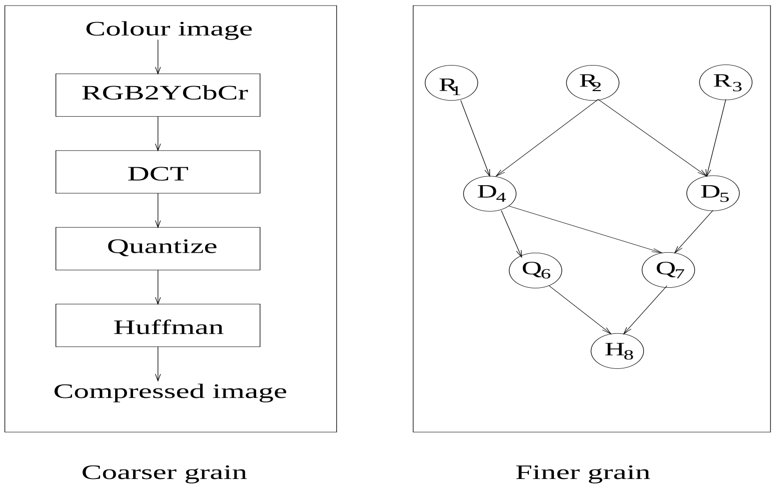

Figure 11. Task graph for jpeg encoder

	Experiment	Latency (ms)
Coarse-grain graph	HW-SW partitioning (no dynamic reconfiguration)	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)	7.51
	Multiple implementation points	6.82
	Best implementation points	9.58

Table 6. Schedule length for different HW-SW partitioning of JPEG encoder

Numerical data on the significant reconfiguration time for a CLB column confirms observations from previous researchers [1] 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 $256X256$ colour image.

Table 6 presents a summary of schedule length estimates (in ms) we generated from various experiments. The first row (**16.74ms**) 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 row (**9.9ms**) 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 (**9.04ms**), 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 (**7.21ms**) represents the results generated by our heuristic on the finer-grain graph- this is optimal for this representation.

Experiment on heterogeneity

For the next experiment in the fifth row (**6.82ms**) we considered that the resource constraint of 8 columns now included one specialized resource (heterogenous) column, i,e, the new resource constraint was a set of 7 CLB columns and 1 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.

Best implementation points only

For the final experiment in row 6 (**9.58ms**) 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 a 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.

7 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 bsaed on ILP (integer linear programming). With the help of this exact approach, we demonstrated that ignoring linear task placement constraints imposed by partial dynamic reconfiguration can result in optimal, but physically unrealizable schedules. Unlike existing ILP-based approaches to HW-SW partitioning, our formulation simultaneously places tasks while scheduling – it also considers the key feature of configuration pre-fetch 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 Kernighan-Lin/Fiduccia-Matheyes 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 run-time 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. Also, our heuristic currently is focused on homogenous implementations. In the future, we will focus on issues leading to high-quality solutions in heterogenous scenarios.

8 Acknowledgements

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

References

[1] J. Noguera, R. M. Badia, ”Power-Performance trade-offs for reconfigurable computing”, CODES+ISSS, 2004

[2] P-H Yuh, C-L Yang, Y-W Chang, H-L Chen, ”Temporal floorplanning using the T-tree formulation”, ICCAD, 2004

[3] S. Ghiasi, M. Sarrafzadeh, ”Optimal Reconfiguration Sequence Management”, ASPDAC, 2003.

[4] Z.Li, ”Configuration management techniques for reconfigurable computing”, Ph.D. Thesis, Northwestern University, 2002

[5] K Ben Chehida, M Auguin, ”HW/SW partitioning approach for reconfigurable system design”, CASES 2002

[6] J. L. Ramirez-Alfonsin, B. A. Reed (Eds.), ”Perfect Graphs”, John Wiley and Sons, 2001.

[7] S.P. Fekete, E.Kohler, J.Teich, ”Optimal FPGA module placement with temporal precedence constraints”, DATE, 2001

[8] H. Singh, G. Lu, E. M. C. Filho, R. Maestre, M-H. Lee, F. J. Kurdahi, N. Bagherzadeh, ”MorphoSys: case study of a reconfigurable computing system targeting multimedia applications”, DAC, 2000.

[9] B. Mei, P. Schaumont, S. Vernalde, ”A hardware-Software Partitioning and scheduling algorithm for dynamically reconfigurable embedded systems”, ProRisc workshop on Ckts, Systems and Signal processing, Nov 2000.

[10] B. Jeong, S. Yoo, S. Lee, K. Choi, ”Hardware-Software Cosynthesis for Run-time Incrementally Reconfigurable FPGAs”, ASPDAC, 2000.

[11] K. S. Chatha, R. Vemuri, ”An iterative algorithm for Hardware-Software partitioning, Hardware design Space Exploration, and scheduling”, Jrnl Design Automation for Embedded Systems, V-5, 2000

[12] M Kaul, R Vemuri, ”Optimal temporal partitioning and Synthesis for reconfigurable architectures”, DATE 1998

[13] S. Hauck, ”Configuration pre-fetch for single context reconfigurable processors”, FPGA, 1998.

[14] R P Dick, D L Rhodes, W Wolf, ”TGFF: task graphs for free”, CODES 1998

[15] F. Vahid, T. D. Le, ”Extending the Kernighan-Lin heuristic for Hardware and Software functional partitioning”, Jrnl Design Automation for Embedded Systems, V-2, 1997

[16] M. J. Wirthlin, ”Improving functional density through Run-time Circuit Reconfiguration”, PhD Thesis, Electrical and Computer Engineering Dept, Brigham Young Univesity, 1997.

[17] R Niemann, P Marwedel, ”An Algorithm for Hardware/Software Partitioning using mixed Integer Linear Programming”, Jrnl Design Automation for Embedded Systems, 1997

[18] H. Murata, K. Fujiyoshi, S. Nakatake, Y. Kajitani, ”Rectangle-packing based module placement”, ICCAD, 1995

[19] P.Hansen, B. Jaumard, V. Mathon, ”Constrained Non-linear 0-1 programming”, ORSA Journal of Computing, Vol 5, No 2, 1993

[20] J.E. Beasley, ”An exact two-dimensional non-guillotine cutting tree search procedure”, Op. Researchm V-33, 1985

[21] C. M. Fiduccia, R. M. Mattheyes, ”A Linear-time heuristic for improving network partitions”, DAC, 1982

[22] B Kernighan, S Lin, ”An efficient heuristic procedure for partitioning graphs”, The Bell System Technical Journal, V-29, 1970

[23] W L Winston, M Venkataraman, ”Introduction to Mathematical Programming”, Thomson Brooks Cole Publishers, 4’th edition, 2003

[24] M. Sarrafzadeh, C. K. Wong, ”An Introduction to VLSI Physical Design” McGraw Hill, 1994.

[25] www.xilinx.com