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

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

ABSTRACT

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

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

1. INTRODUCTION

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

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

This work makes several contributions:

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

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

2. RELATED WORK

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

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

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

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

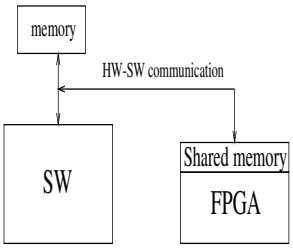

Figure 1: System architecture

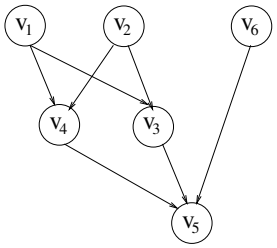

Figure 2: Dependency task graph

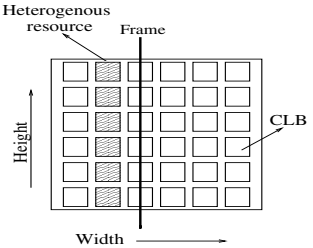

Figure 3: Heterogenous FPGA with partial RTR

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

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

3. PROBLEM DESCRIPTION

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

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

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

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

Dynamically reconfigurable FPGA

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

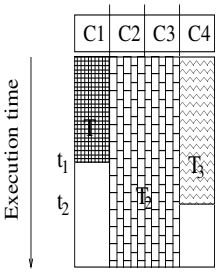

Figure 4: Simple infeasible

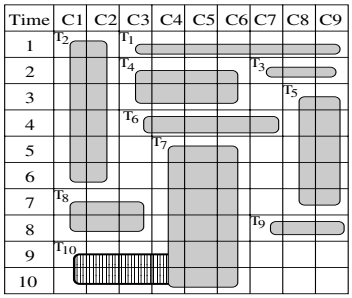

Figure 5: Detailed infeasible

4. KEY ISSUES IN SCHEDULING

4.1 Criticality of linear task placement

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

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

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

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

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

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

4.2 Heterogeneous implementations

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

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

4.3 Scheduling for configuration prefetch

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

5. PROPOSED APPROACH

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

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

5.1 Heuristic formulation

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

Code segment 1: KLFM loop

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

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

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

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

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

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

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 6: Task parameters

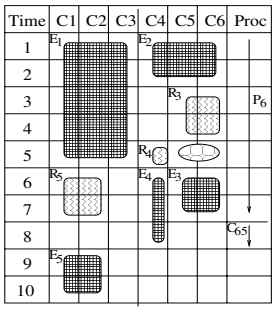

Figure 7: Optimally placed

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:

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

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.

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 2, and its associated parameters in Figure 6. 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 7. 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 7, 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 [13].

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 2** we present our approach to EST computation that addresses the issues we discussed above. 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 7 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.

Code segment 2: 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
```

5.2 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 do this by simply adding a type descriptor for each column in our resource description. Thus all resource queries at a time instant check the column type descriptor while looking for free space at that instant. Some simple initial preprocessing makes the searches more efficient.

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. In this section we present a reasonable subset of our experiments– additional details are in [16]. 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 ca-

	Placement Unaware		Placement Aware	
Test	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 1: Feasibility results and heuristic quality for small tests

pability. 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.

Experimental setup

The following assumptions form the basis of our numerical data:

HW Device: similar to Xilinx Xc2V2000, organized as a CLB matrix of 56 rows and 48 columns.
SW execution unit: PowerPC operating at 300 MHz.
Communication bus: 64-bit PLB operating at 133 MHz.
1 CLB = 22 frames (a total of 1456 frames on the entire device);
Total reconfiguration time = 17.01 ms (SelectMAP port at 50 Mhz);
Maximum suggested reconfiguration frequency = 66 MHz.
reconfiguration overhead for task occupying one CLB column=
22/1456 * 17.01 * 50/66 = 0.19 ms

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.

Experiments on feasibility

Table 1 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 Table1, 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) [7]. As Table1 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.

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 1. The data indicates that for the small cases, T_{heu} corresponds to schedules that are reasonably close in quality to the exact solution.

Test group	Few cols (8,12)	More Cols (16,20)	Avg gain	Run-time (s) (20 cols)
v20	1.83%	7.60%	4.71%	.02
v40	1.68%	7.83%	4.75%	2.0
v60	4.93%	8.80%	6.86%	22
v80	4.09%	10.57%	7.33%	90
v100	8.96%	11.92%	10.44%	180
Avg gain	4.3%	9.34%		

Table 2: Aggregate improvements in schedule length

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 [10]. In these tests, each task had a single homogenous implementation point. In subsequent discussions, $v20$, $v80$, etc, denote sets of graphs that have approximately 20 nodes, 80 nodes, etc. 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.

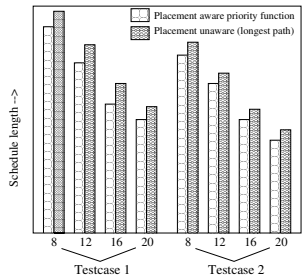

Figure 8: Sample experiments for v60

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: 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 8 we present a sample of the tests we conducted. For two test graphs in set $v20$ 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 8 shows that our placement-aware priority function *consistently* generates better schedules. Table 2 summarizes the result for 60 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 6.86%. 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 2, 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 graph size and increase in available resources. The key difference is that LPF also tries to improve schedule length by

A	B	C	D	E	F
16.74	9.9	9.04	7.21	6.82	9.58

Table 3: Schedule length for different HW-SW partitioning of JPEG encoder

prefetch, but only after selecting the task to be scheduled, while our heuristic considers placement implications in task selection.

Case study of JPEG encoder

We next conducted a detailed analysis for the JPEG encoding algorithm 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 was 11 columns. Our schedule length data is for processing a block corresponding to a $256X256$ colour image under a resource constraint of 8 columns.

Table 3 presents a summary of schedule length estimates (in ms) we generated from various experiments. The first 3 columns A,B,C correspond to experimental data for the given task graph. Column A (**16.74ms**) represents the first experiment with static HW-SW partitioning (without considering the dynamic nature of HW), B (**9.9ms**) represents HW-SW partitioning when the HW is partially reconfigurable at run-time, C (**9.04ms**) corresponds to schedule improvement with optimal prefetch.

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 in columns D, E, F corresponds to experimental data for the finer-grain task graph. Column D (**7.21ms**) represents the results generated by our heuristic on the finer-grain graph- this is optimal for this representation.

Heterogenous Architecture

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

While evaluating the schedule length improvement in Column E, a key factor to be noted is our realistic assumption about the reconfiguration overhead for a specialized resource column- on the Virtex-II, a resource column has 64 frames whereas a CLB column has 22 frames leading to a significantly higher reconfiguration overhead for a specialized column. This fact leads to significantly less speedup than would be expected simply from considering execution time difference between homogenous and heterogenous implementations. Another important observation was that heterogeneity restricts placement significantly and the relative location of the specialized resource column strongly affects the schedule length.

Column F (**9.58ms**) was our final experiment where 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.

Run time of algorithm: Table 2 also shows the average run-time of our approach (in seconds) for 20–100 tasks given an area-constraint of 20 columns– 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.

7. CONCLUSION

In this paper, we first demonstrated with an exact approach that ignoring linear task placement constraints imposed by a reconfigurable architecture with partial dynamic reconfiguration can result in optimal, but physically unrealizable schedules. We next proposed a placement-aware HW-SW partitioning heuristic that simultaneously partitions, schedules and does linear placement of tasks on such devices. Our approach considers the key issues of configuration prefetch, the bottleneck of a single reconfiguration controller. We conducted a wide range of experiments to validate the quality of solutions generated by our placement-aware heuristic. Placement and consideration of multiple implementations in partitioning make it easy to extend our approach to heterogenous FPGAs. We demonstrate with a case study the exploratory capabilities provided by our approach. 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- more investigations are required into issues leading to high-quality solutions in heterogenous scenarios.

8. REFERENCES

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

[2] S. Ghiasi, M. Sarrafzadeh, "Optimal Reconfiguration Sequence Management", ASPDAC, 2003.

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

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

[5] 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.

[6] 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.

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

[8] 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

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

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

[11] 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

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

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

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

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

[16] S Banerjee, E Bozorgzadeh, N Dutt, "HW-SW partitioning for architectures with partial dynamic reconfiguration", Technical Report CECS-TR-05-02, UC Irvine.

[17] www.xilinx.com