Temporal Partitioning combined with Design Space Exploration for Latency Minimization of Run-Time Reconfigured Designs [*]

Meenakshi Kaul
mkaul@ececs.uc.edu

Ranga Vemuri
Ranga.Vemuri@UC.EDU

Laboratory for Digital Design Environments, Department of ECECS
University of Cincinnati, Cincinnati, OH 45221–0030

Abstract

We present combined temporal partitioning and design space exploration techniques for synthesizing behavioral specifications for run-time reconfigurable processors. Design space exploration involves selecting a design point for each task from a set of design points for that task to achieve latency minimization of partitioned solutions. We present an iterative search procedure that uses a core ILP (Integer Linear Programming) technique, to obtain constraint satisfying solutions. The search procedure explores different regions of the design space while accomplishing combined partitioning and design space exploration. A case study of the DCT (Discrete Cosine Transform) demonstrates the effectiveness of our approach.

1 Introduction

Run-time reconfigurable processors [12, 22] are becoming increasingly viable with the advent of modern field-programmable devices. Run-time reconfiguration allows large circuits to be implemented by multiple configurations of the same processor. This necessitates temporal partitioning of the specification. Temporal partitioning involves dividing the specification into multiple segments, that execute one after the other on the reconfigurable processor. The effectiveness of temporal partitioning of applications has been presented in [1, 2]; the partitioning, however, is accomplished manually. Therefore, techniques to automatically partition designs temporally are needed.

The design space of a design consists of many alternative design implementations which vary in area and latency. In order to explore the design space, it is important to integrate partitioning and synthesis techniques. An optimally synthesized design for constrained architectures, should meet the area constraint while achieving the design whose execution time is the least over all the possible design alternatives. Therefore in this work we focus on methods to reduce the latency of the partitioned design, while meeting the architectural constraints. *We present an iterative refinement procedure that iteratively explores different regions of the design space to yield latency-reduction of the partitioned design. An Integer Linear Programming (ILP) based integrated temporal partitioning and design space exploration technique forms a core solution method which is used to explore different regions in a latency directed search heuristic. We demonstrate the effectiveness of our technique with experimental results.*

Currently, tools for temporal partitioning rely on the manual specification of the partitioning points [4]. Others have have extended existing scheduling and clustering techniques of high-level synthesis [3, 5, 11]. In [5, 11] the temporal partitioning technique involves partitioning gate-level designs. Since the design to be partitioned is already synthesized, different synthesis options for achieving lesser latency partitioned solutions cannot be explored. Our technique can simultaneously handle multiple design constraints, eg., FPGA resources, on-board memories, and perform design exploration that cannot be handled by current techniques in [3, 5, 11]. In an earlier work [8], we presented a mathematical model for combined temporal partitioning and operation level synthesis. The number of alternative solutions explored becomes very large and the technique can be used to synthesize small-scale behavior specifications.

[14] presents an extended bi-partitioning problem for co-design where partitioning and design point selection is performed sequentially, unlike our combined approach. Simultaneous spatial partitioning and synthesis is formulated as an ILP in [9]. [13] presents an ILP-based methodology for hardware software partitioning of co-design systems. Resource constrained scheduling and binding at operation level for ASICs has been formulated as an ILP in [10]. Here, the latency of a design is the number of control steps in the solution. A control step would correspond to a temporal partition in the temporal partition formulation if the latency of each partition is a constant. However, since the latency of each temporal partition will be different based on the tasks mapped to the partition, we need to generate formulation to determine the latency of the critical path mapped to a partition. And unlike in scheduling since memory is a constraint, our model formulates memory constraint equations.

There exist no automated temporal partitioning techniques that explore different design-space points for the

[*] This work is supported in part by the US Air Force, Wright Laboratory, WPAFB, under contract number F33615-97-C-1043.

design. *In this paper we demonstrate how such integrated temporal partitioning and design space exploration techniques can be used successfully to obtain near-minimal latency designs.*

2 Motivation

Temporal partitioning can be performed at various stages in the design process. It can be performed at RT-level or gate-level after the design has been synthesized [11]. The effect of partitioning should influence the exploration of the synthesis design space, but this information would be unavailable while the design is being synthesized. Alternatively, behavioral temporal partitioning can be performed first, and the resulting partitions can be synthesized. However, the synthesis cost of a design on the given architecture will determine the partitioning of the design; therefore it is very important to explore the design space while performing synthesis and partitioning together.

Growing design complexity has lead designers to generate designs at higher levels of abstraction, such as, the behavior level. In this paper, we concentrate on behavior level design descriptions to be temporally partitioned. We assume the input specification to be a task graph, where each task consists of a set of operations. Task boundaries can be given by the designer, or tasks can be automatically derived from the behavior specification by clustering or template extraction techniques [15].

Design Alternatives for Tasks : Depending on the resource/area constraint for the design different implementations of the same task, which represent different area-time tradeoff points, can be contemplated. These different implementations are *design points/Pareto points* [19] in the design space of a task. If a task is implemented with less resources, then the operations in the task will be executed serially, thus increasing the latency of the task. On the other hand, an implementation with more resources reduces the latency but increases the area. Therefore, choosing the best design point for each task may not necessarily result in the best overall design for the specification. The most optimal design point for a task will depend on the architectural constraints and the dependency constraints among the tasks. We make use of a high-level synthesis estimation tool [17] for obtaining various design points for a task. In the subsequent discussion, we will express the latency of a design point in terms of total execution time and not in number of clock cycles.

If the number of design alternatives for a task are too many, then exploring the large design space can become too computationally expensive. In such cases, 'candidate' design points must be obtained by effective design space pruning techniques [15]. Since there is a gap between the behavior description and the final synthesized design, it is important that we have accurate synthesis estimates for the tasks. We can use more sophisticated High-Level Synthesis estimators which incorporate layout estimation techniques. Such partitioned designs, can then be predictably taken down to the actual FPGA layout [16].

Area-Latency Tradeoff : The design alternatives or solutions for a specification to be temporally partitioned will vary in the number of temporal partitions and the latency of the partitioned design. For the *spatial* partitioning problem, increasing the number of partitions has the effect of increasing the overall area for the design, and directly affects the latency of the design. Increasing the area generally increases the number of operations that can execute in parallel (if no dependency constraints exist) and thus decreases the latency of the design. However, for a temporal partitioning system increasing the number of partitions increases the area available for the design, but this increase is 'over time' and not 'over space'. This increase in number of partitions may or may not result in the reduction of the latency of the design. To understand this effect, we need to consider the kind of reconfiguration time overhead of the different reconfigurable architectures -

• The reconfiguration time is orders of magnitude greater than the task graph latency. The *Wildforce* reconfigurable board [22] with reconfiguration time in milli-seconds is is an example.

• The reconfiguration time is comparable to the latency of task graphs. An example is the *Time-Multiplexed FPGA* in [12] with reconfiguration time in nano-seconds.

When the reconfiguration overhead is very large compared to the execution time of the task, it is clear that minimizing the number of temporal partitions will achieve the *smallest latency* in the overall design. However when the reconfiguration overhead is small, minimizing the number of partitions may not minimize the overall latency for the design. As we have less partitions, lesser FPGA resources are available. Therefore to minimize the number of partitions the temporal partitioning system will pick among the available design space, those design points which have least area. These design points will, however, have more latency. For example, consider that the difference in latency between two design points for a task is 100 ns, and the reconfiguration time is 30 ns. The optimal solution may be found by increasing the number of partitions over which the design is partitioned, so that the design point with larger area is able to fit, with a reduction of 70 ns over the overall latency of the design.

Usually partitioning tools focus on finding a solution for a given partition size (k-way partitioning, where k is an input to the system). Our temporal partitioning tool performs three functions *(1) maps tasks to partitions, (2) maps each task to appropriate design point, and (3) explores multiple partitioning solutions, so that an appropriate partition size is achieved, and the latency of the partitioned solution is reduced*.

3 Temporal Partitioning System

The inputs to our temporal partitioning and design space exploration system are - (1) Behavior specifications (2) Target Architecture Parameters. In formal notation, the inputs are stated as -

T	*set of tasks in the task graph.*
$t_i \rightarrow t_j$	*a directed edge between tasks, $t_i, t_j \in T$, exists in the task graph.*
$B(t_i, t_j)$	*number of data units to be communicated between tasks t_i and t_j.*
$B(env, t_j)$	*number of data units to be read by task t_j from the environment.*
$B(t_i, env)$	*number of data units to be written from task t_i to the environment.*
R_{max}	*resource capacity of the reconfigurable processor.*
M_{max}	*temporary on-board memory size.*
C_T	*reconfiguration time of the reconfigurable processor.*

The behavior specifications are in the form of a directed graph called the *Task Graph*. The vertices in the graph denote tasks, and the edges denote the dependency among tasks. Data communicated between two tasks, $B(t_i, t_j)$, will have to be stored in the on-board memory of the processor, if the two tasks connected by an edge are placed in different temporal partitions. Data being communicated to the tasks, $B(env, t_j)$, or from the tasks, $B(t_i, env)$ to the host also needs to be stored. The target architecture parameters specify the underlying resources and the reconfiguration time, C_T, for the device. Typically, resource capacity, R_{max}, is the combinational logic blocks/function generators on the FPGAs of the reconfigurable device. M_{max}, is the memory for storage of intermediate data available on the reconfigurable processor.

3.1 Preprocessing

Design Point Generation: Each task in the task graph is synthesized by a high level synthesis estimation tool. The high level synthesis tool generates a set of *design points* for each task. Each design point has an associated *module set*[18]. A module set, m, consists of the set of, possibly multiple, functional units used to implement the design point. Each design point is characterized by its area and latency. Each task t, will have a set of module sets, M_t, corresponding to the set of synthesized design points. We state this formally as -

M_t	*set of module sets for a task $t \in T$.*
$R(m)$	*area for a design point using module set $m \in M_t$.*
$D(m)$	*latency of a design point using module set $m \in M_t$.*

Partition bounds Estimation: To find the number of partitions over which the temporal partitioning solution should be explored we calculate two bounds -
1. MinAreaPartitions(): For calculating the lower bound on number of partitions N^l_{min} we sum the *minimum* area module set, m, for each task. This value divided by the FPGA area will be the minimum number of partitions required to obtain a solution.

$$N^l_{min} = \sum_{t \in T} R(m)/R_{max}, \quad \{m \mid \forall m \in M_t, min(R(m))\}$$

2. MaxAreaPartitions(): Ideally, we should be able to establish an upper bound on the number of partitions needed to be explored by the partitioner, if the maximum area design point for each task is chosen. However, we cannot establish an upper bound on the maximum number of partitions. If a task is too large to fit in some temporal partition, it must go to a later partition. Then all the descendents of this task also cannot occupy the earlier temporal partition. This will leave some area on temporal partitions unoccupied due to dependency constraints, and the task graph will not fit even though there is enough area left unoccupied on the partitions. We could have established an upper bound on the maximum number of partitions to be equal to the number of tasks in the task graph. However, this is a very pessimistic bound and usually so many partitions need not be explored. We define, the minimum number of partitions, N^u_{min}, that need to be explored if the *maximum* area design point for each task is mapped by the partitioner, to be -

$$N^u_{min} = \sum_{t \in T} R(m)/R_{max}, \quad \{m \mid \forall m \in M_t, max(R(m))\}$$

Latency bounds Calculation: The latency of a synthesized design will involve two components - (1) latency due to the actual execution of the tasks in the task graph, (2) latency due to the reconfiguration overhead. For a given number of temporal partitions, N, we can calculate the upper and lower bounds on the latency of the design.
1. *MaxLatency(N):* The worst latency D_{max}, will occur when all tasks are serially executed. For latency calculation, we will use the design point with maximum latency for each task. This latency added to the reconfiguration overhead will be the upper bound latency for N partitions.

$$D_{max} = \sum_{t \in T} D(m) + N * C_T$$

2. *MinLatency(N):* For obtaining the lower bound we consider for each task the fastest (minimum latency) design point. We obtain the latency for all the paths in the task graph, by summing up the minimum latency of the tasks along each path. The maximum latency value over all such path latencies in the task graph, gives us the lower bound on the latency. This added to the reconfiguration overhead will be the lower bound latency for N partitions.

$$D_{min} = max \; \{ \text{ all least latency paths in T } \} \; + N * C_T$$

3.2 Algorithm

Informally, the algorithm has the following steps -
1. Obtain a constraint satisfying solution for the starting partition size N^l_{min}, and latency constraints D_{max}, D_{min} for this partition bound.
2. Find lower latency solutions by progressively exploring different regions of the search space, by tightening the latency constraints, for the current partition bound.
3. Increase the partition size bound and go to step 2.

```
Algorithm $Reduce\_Latency(N, D_{max}, D_{min})$
begin
  $D_a \leftarrow 0$
  FormModel()
  if SolveModel() = $Infeasible$
    return($D_a$)
  $D_a \leftarrow$ CalculateSolnLatency() /* Achieved latency of solution */
  while ($D_{max} - D_{min} \geq \delta$) and ($D_a - D_{min} \geq \delta$)
    $D'_{max} = D_{max}$
    /* Binary subdivision of achievable latency range */
    $D_{max} = (D_{max} + D_{min})/2$
    while ($D_{max} \geq D_a$)
    /* we have already achieved a latency $D_a$ less than $D_{max}$ */
      $D_{max} = (D_{max} + D_{min})/2$
    end while
    FormModel()
    if SolveModel() = $Infeasible$
      /* tighten lower bound to remove infeasibility */
      $D_{min} = D_{max}$
      $D_{max} = D'_{max}$
    else
      $D_a \leftarrow$ CalculateSolnLatency()
    end if
  end while
  return($D_a$)
end Algorithm $Reduce\_Latency$
```

Figure 1. Latency Refinement Procedure

3.2.1 Exploration by latency constraint reduction

Figure 1, describes formally the latency reduction algorithm. It is an iterative procedure that obtains near-optimal latency solutions for a given partition bound, N, and latency bounds D_{max} and D_{min}. It finds a constraint satisfying solution between D_{max} and D_{min}. Once a solution is obtained, the upper bound on latency is reduced to $(D_{max} + D_{min})/2$, and a new solution for these constraints is found. If a feasible solution is obtained, then the obtained latency of the solution becomes the upper bound for a new search. If no feasible solution is obtained, then this latency becomes the new lower bound. It continues this binary subdivision on the latency bounds, till the difference between the upper and lower bounds becomes very small, or no more feasible solutions are found. The tolerable difference between the lower and upper latency bounds is a user defined parameter, δ, called the *latency tolerance*. Latency tolerance defines how much of the design space can be left unexplored in one run of the algorithm. If the tolerance is small, more iterations will be spent in obtaining a solution, thus increasing the run time. If a large run time is not acceptable then latency tolerance can be increased. In practice, we can set the latency tolerance to a small percentage of the $MaxLatency(N)$ of the task graph.

We have formulated the temporal partitioning and design space exploration problem as an ILP (presented in Section 3.2.3). We do not use the ILP for finding optimal solutions, but instead use it to obtain a feasible solution for a problem. Our latency reduction procedure then makes the constraints tighter, reformulates the ILP and solves it for the new problem. We could have also used the same ILP to solve the problem to optimality, but we found that this is feasible only for small problem sizes of up to 10 tasks. For larger designs, therefore we have developed this directed search procedure, which reduces the search space for *each run* of the ILP solver, while still exploring the whole design space over all iterations. This claim has been substantiated in Section 4 by demonstrating that for small designs the solution obtained by this procedure and an ILP solved to optimality is the same. In the algorithm, procedure $FormModel()$ forms the ILP model, $SolveModel()$ then solves the model and returns with the first feasible constraint satisfying solution.

```
Algorithm $Refine\_Partition\_Bound()$
begin
  $N^u_{min} \leftarrow$ MaxAreaPartitions()
  $N^l_{min} \leftarrow$ MinAreaPartitions()
  $N \leftarrow N^l_{min} + \alpha$ /* starting partition number */
  $D_{max} \leftarrow$ MaxLatency(N)
  $D_{min} \leftarrow$ MinLatency(N)
  $D_a \leftarrow Reduce\_Latency(N, D_{max}, D_{min})$
  while ($D_a$ = 0) /* Partition bound was infeasible */
    $N \leftarrow N + 1$ /* next partition number */
    $D_{max} \leftarrow$ MaxLatency(N)
    $D_{min} \leftarrow$ MinLatency(N)
    $D_a \leftarrow Reduce\_Latency(N, D_{max}, D_{min})$
  end while
  while $N < N^u_{min} + \gamma$ and not TimeExpired()
    $N \leftarrow N + 1$ /* Relax N */
    $D_{min} \leftarrow$ MinLatency(N)
    if $D_{min} \geq D_a$
      return($D_a$) /* This is the best solution */
      else
      /* find a better solution by taking $D_a$ as upper bound */
      $D'_a \leftarrow Reduce\_Latency(N, D_a, D_{min})$
      if $D'_a \neq 0$ /* Feasible */
        $D_a \leftarrow D'_a$
      end if
    end if
  while
  return($D_a$) /* return with the last known best solution */
end Algorithm $Refine\_Partition\_Bound$
```

Figure 2. Partition Refinement Procedure

3.2.2 Partition Space Exploration

To explore better solutions for the temporal partitioning problem, we need to explore more than one partition bound. Finding the ideal partition size, N, is also an iterative procedure, shown in Figure 2. We calculate the minimum number of partitions, N^l_{min}, as described earlier. We can start from this partition number or from a slight relaxation, defined by α, called the *Starting Partition Relaxation*. We start the search at $N^l_{min} + \alpha$ and obtain a near optimal solution, by using the Algorithm $Reduce\_Latency$ as explained earlier. The resultant latency is the achieved latency D_a for N partitions. To explore better solutions, we now relax N by 1, and call $Reduce\_Latency$ again. This time however, since we are looking for a better solution than the one we have already achieved, D_a is the latency upper bound for the new search by $Reduce\_Latency$. We continue to relax N and look for better solutions until the value of N reaches

$N^u_{min} + \gamma$. Here γ is a user controlled parameter, called the *Ending Partition Relaxation*, which defines the number of partitions beyond N^u_{min} that must be explored while searching for better solutions.

The partitions ranging from $N^l_{min} + \alpha$ to $N^u_{min} + \gamma$ need to be explored for searching the whole design space. We have introduced these parameters, so that a user can direct the partition space search if the user has more knowledge of the solution to the problem. We give an example of how this can be done. Using a heuristic, if we map the least area design points for each task we arrive at a solution with partition size $N^{'}$. If $N^{'}$ is greater than N^l_{min} then $\alpha = N^{'} - N^l_{min}$. (However this is not a true lower bound as the heuristic may have missed a solution with lesser number of partitions than $N^{'}$.) Similarly, using a heuristic and mapping the maximum area design point for each task, we arrive at a solution with $N^{''}$ partitions. This is an upper bound on the partition size. If $N^{''} > N^u_{min}$, then $\gamma = N^{''} - N^u_{min}$. We are currently studying how to achieve tighter upper and lower bounds for partition size, and incorporating them automatically in our algorithm. However the facility of giving α and γ will still be provided to the user. Intuitively, for reconfigurable architectures with large reconfiguration overhead, since the minimum latency solution will be obtained in the least number of partitions, α & $\gamma = 0$.

3.2.3 ILP formulation

We build the temporal partitioning model for the given inputs and the values of N, D_{max} and D_{min} derived from the algorithms describe earlier. After linearization of the non-linear constraints, we solve it using a linear programming solver. To state formally the mathematical model we use the following definitions

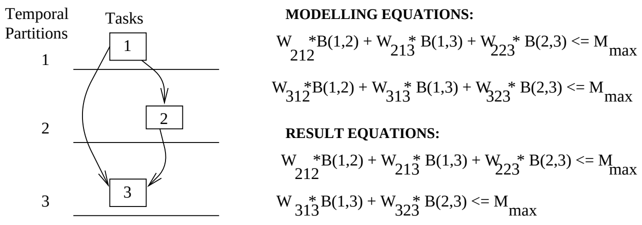
T_l	*set of tasks $t_i \in T$, where $\forall t_j \in T, \neg(t_i \rightarrow t_j)$, (leaf tasks of T).*
T_r	*set of tasks $t_j \in T$, where $\forall t_i \in T, \neg(t_i \rightarrow t_j)$, (root tasks of T).*
$t_i \xrightarrow{p} t_j$	*a directed path from $t_i \in T$ to $t_j \in T$.*
$P_{l \xrightarrow{p} r}$	$\{ t_i \xrightarrow{p} t_j \mid (t_i \in T_r) \wedge (t_j \in T_l)\}$, *(set of paths from root tasks to leaf tasks).*

Variables and Constraints : Variable y_{tpm}, models partitioning and design point selection for a task. $w_{pt_1t_2}$, models data transfer requirement across partition boundaries. η, is the actual number of partitions finally used in the solution and will be less than or equal to N. d_p, models the execution time of a temporal partition.

$$y_{tpm} = \begin{cases} 1 & \text{if task } t \in T \text{ is placed in partition p,} \\ & \quad 1 \leq p \leq \text{N, using module set } m \in M_t \\ 0 & \text{otherwise} \end{cases}$$

$$w_{pt_1t_2} = \begin{cases} 1 & \text{if task } t_1 \text{ is placed in any partition } 1 \cdots p-1 \text{ and} \\ & \quad t_2 \text{ is placed in any of } p \cdots N \text{ and } t_1 \rightarrow t_2 \\ 1 & \text{if task } t_1 \text{ is placed in partition } p \text{ and } t_2 \text{ is} \\ & \quad \text{placed in any of } p+1 \cdots N \text{ and } t_1 \rightarrow t_2 \\ 0 & \text{otherwise} \end{cases}$$

$\eta =$ Number of partitions actually used in solution.



MODELLING EQUATIONS:

$W_{212}*B(1,2) + W_{213}*B(1,3) + W_{223}*B(2,3) <= M_{max}$

$W_{312}*B(1,2) + W_{313}*B(1,3) + W_{323}*B(2,3) <= M_{max}$

RESULT EQUATIONS:

$W_{212}*B(1,2) + W_{213}*B(1,3) + W_{223}*B(2,3) <= M_{max}$

$W_{313}*B(1,3) + W_{323}*B(2,3) <= M_{max}$

Figure 3. The constraints to be satisfied if tasks are mapped to partitions as shown

$d_p =$ latency of partition p.

Variables y_{tpm}, $w_{pt_1t_2}$ are 0-1 variables, η is an integer variable and d_p can be integer or real depending on whether the latency values are integer or real.

Uniqueness Constraint: Each task should be placed in exactly one partition among the N temporal partitions, while selecting one among the various module sets for the task.

$$\forall t \in T \quad : \sum_{m \in M_t} \sum_{p=1}^{N} y_{tpm} = 1 \qquad (1)$$

Temporal order Constraint: Because we are partitioning over time, a task t_1 on which another task t_2 is dependent cannot be placed in a later partition than the partition in which task t_2 is placed. It has to be placed either in the same partition as t_2 or in an earlier one.

$$\forall t_2,\ \forall t_1 \rightarrow t_2,\ \forall p_2,\ 1 \leq p_2 \leq N-1 \quad :$$
$$\sum_{m_1 \in M_{t_1}} \sum_{p_2 < p_1 \leq N} y_{t_1p_1m_1} + \sum_{m_2 \in M_{t_2}} y_{t_2p_2m_2} \leq 1 \qquad (2)$$

Memory Constraint: Data transfer across partition boundaries will occur due to two dependent tasks being placed in different temporal partitions. This intermediate data needs to be stored between partitions and should be less than the memory, M_{max}, of the reconfigurable processor. The variable $w_{pt_1t_2}$, if 1, signifies that t_1 and t_2 have a data dependency and are being placed across temporal partition p. Therefore the data being communicated between them, $B(t_1, t_2)$, will have to be stored in the memory of partition p. The sum of all the data being communicated across a partition should be less than the memory constraint of the partition.

$$\forall p,\ 1 \leq p \leq N \quad : \sum_{t \in T} \sum_{p \leq p_2 \leq N} (y_{tp_2} * B(env, t)) +$$
$$\sum_{t \in T} \sum_{1 \leq p_3 \leq p} (y_{tp_3} * B(t, env)) +$$
$$\sum_{t_2 \in T} \sum_{t_1 \rightarrow t_2} (w_{pt_1t_2} * B(t_1, t_2)) \leq M_{max} \qquad (3)$$

Note that the variable $w_{pt_1t_2}$ has to model communication among tasks which are both on adjacent and non-adjacent temporal partitions. In Figure 3, we show how this variable models data transfer. We show in the figure the original equations used to model the constraints in the example for Temporal Partitions 2 and 3. The result equations show the variables which will be 1 in the mapping of tasks to partitions shown in the example and the constraint which has to be satisfied. $w_{pt_1t_2}$ are 0-1 non-linear terms constrained as -

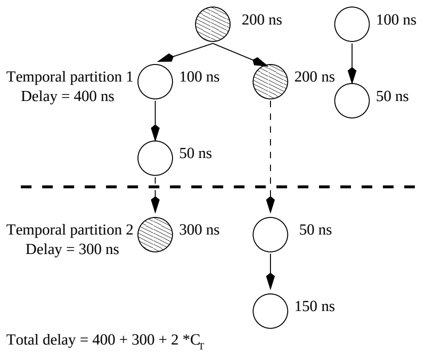


Figure 4. **Latency Estimation**

$$\forall p,\ 1 \le p \le N,\ \forall t_2 \in T,\ \forall t_1 \to t_2, \quad : \qquad w_{pt_1t_2} \ge \textstyle\sum_{1 \le p_1 < p} y_{t_1p_1} * \sum_{p \le p_2 \le N} y_{t_2p_2} \tag{4}$$

$$\forall p,\ 1 \le p \le N,\ \forall t_2 \in T,\ \forall t_1 \to t_2, \quad : \qquad w_{pt_1t_2} \ge y_{t_1p} * \textstyle\sum_{p+1 \le p_2 \le N} y_{t_2p_2} \tag{5}$$

Equations (4) and (5) are non-linear. We can use linearization techniques to transform the non-linear equations into linear ones, so that the model can be solved by a Linear Program solver. Linearization techniques have been used successfully before in [8] to solve the combined temporal partitioning and synthesis problem.

Resource Constraint: The sum of area costs of all the tasks mapped to a temporal partition must be less than the overall resource constraint of the reconfigurable processor. Typical FPGA resources include function generators, configurable logic blocks etc. Similar equations can be added if multiple resource types exist in the FPGA.

$$\forall p,\ 1 \le p \le N \quad : \qquad \textstyle\sum_{m \in M_t} \sum_{t \in T} (y_{tpm} * R(m)) \le R_{max} \tag{6}$$

Latency Constraint: The latency of design execution on a partition will be the maximum latency among all the paths of the task graph mapped to that partition. In Figure 4, we show how the latency for a partition is determined. The final mapping of tasks to partitions, with the latency value for each task, is shown. In partition 1, three paths are mapped. The latency of this partition is the greatest latency along a path mapped to the partition, i.e., maximum among 350ns, 400ns, 150ns. The maximum latency in partition 2 is 300ns. Formally the latency of design execution on a temporal partition is given as -

$$\forall p,\ 1 \le p \le N,\ \forall (t_i \xrightarrow{p} t_j) \in P_{l \xrightarrow{p} r} \quad : \qquad \textstyle\sum_{m \in M_t} \sum_{t \in t_i \xrightarrow{p} t_j} (y_{tpm} * D(m)) \le d_p \tag{7}$$

All temporal partitions $1 \cdots N$ used in the formulation, may not be used in the final solution, if the tasks can fit in lesser number of partitions. To calculate the actual number of partitions used in the solution, we determine the highest numbered partition used by any leaf level task in the task graph by the following equation -

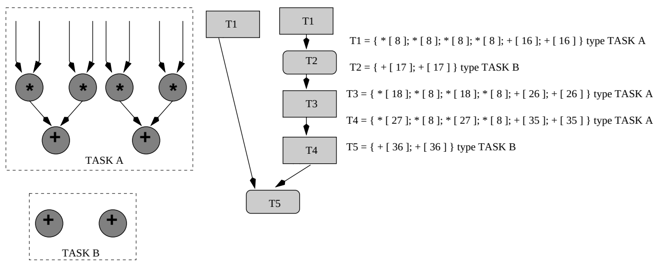

T1
T1
T2
T3
T4
T5
TASK A
TASK B
T1 = { * [8]; * [8]; * [8]; * [8]; + [16]; + [16] } type TASK A
T2 = { + [17]; + [17] } type TASK B
T3 = { * [18]; * [8]; * [18]; * [8]; + [26]; + [26] } type TASK A
T4 = { * [27]; * [8]; * [27]; * [8]; + [35]; + [35] } type TASK A
T5 = { + [36]; + [36] } type TASK B

Figure 5. **Task Graph for the AR Filter**

	Result(Iterative)				Result(Optimal)
N	I	D_{max}	D_{min}	D_a	D_a
3	1	2,745	1,385	Inf.	Inf.
4	1	2,775	1,415	2,150	
	2	2,095	1,415	1,865	
	3	1,755	1,415	Inf.	
	4	1,840	1,755	Inf.	1,865
5	1	1,865	1,445	1,770	
	2	1,650	1,445	Inf.	
	3	1,752	1,650	Inf.	1,770

Table 1. **Temporal Partitioning of the AR filter,** $R_{max} = 196$**,** $C_T = 30ns$**,** $\alpha = 0$**,** $\gamma = 0$**,** $\delta = 30$

$$\forall t \in T_l \quad : \textstyle\sum_{m \in M_t} \sum_{p=1}^{N} (p * y_{tpm}) \le \eta \tag{8}$$

Now the latency constraints can be stated in terms of equations (7) and (8) as -

$$\eta * C_T + \textstyle\sum_{p=1}^{N} d_p \le D_{max} \tag{9}$$

$$\eta * C_T + \textstyle\sum_{p=1}^{N} d_p \ge D_{min} \tag{10}$$

4 Experimental Results

Case Study of AR filter : We present a case study of the Auto Regressive (AR) filter [21]. The size of the task graph is small, but we demonstrate the closeness of the solution obtained by our algorithm and the optimal solution. The task graph for the specification consists of 6 tasks shown in Figure 5. Tasks A and B show the internal structures of the filter tasks. Tasks T1, T3, & T4 have a structure like Task A, but differ in the bit-widths of their operations. Tasks T2 and T5 are like Task B, but again differ in their bit-widths. The bit widths of each operation in each task is also shown in the figure. Due to space limitation, the design points are not shown. Task T1 has three design points, tasks T3 & T4 have two design points each, and tasks T2 and T5 have one design point each. The result of the experimentation is shown in Table 1. N denotes the number of temporal partitions explored. The columns under Result(Iterative) state the result of running our algorithm. I is the iteration of the algorithm, D_{max} and D_{min} are the latency bounds for that iteration. D_a gives the latency of the solution. $Result(Optimal)$ is the result achieved by solving the problem to optimality using the ILP solver. We use $CPLEX$ to solve the ILP problems both for constraint satisfaction and optimal solution. We see that the result of our algorithm matches the optimal solution for this task graph. We have performed a lot of experiments on small task graphs and the solution

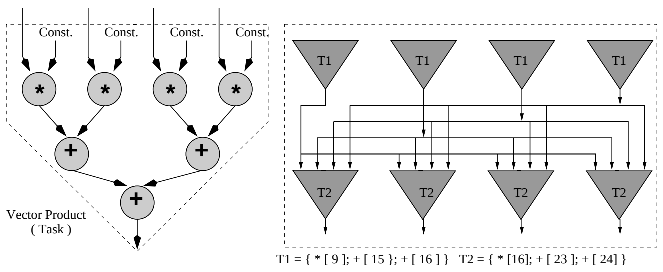

Const.
Const.
Const.
Const.
Vector Product
(Task)
T1
T2
T1 = { * [9]; + [15]; + [16] } T2 = { * [16]; + [23]; + [24] }

Figure 6. Task graph for DCT, 8 of the 32 tasks are shown

Task	D.	Characteristics					
		Area	Latency	$*_9$	$+_{16}$	$*_{16}$	$+_{24}$
T1	1	180	375	4	2		
	2	138	500	2	2		
	3	121	750	1	2		
T2	1	216	420			4	2
	2	188	560			2	2
	3	162	840			1	2

Table 2. Design Points for DCT tasks

for our iterative procedure and an optimally solved ILP has been the same.

Case Study of DCT : For task graphs with larger number of tasks, our iterative constraint satisfaction approach is able to explore in reasonable time more solution space than by solving the problem to optimality. To demonstrate our approach, we undertook a case study of the 4x4 DCT, the most computationally intensive subtask of the $JPEG$ image compression algorithm [20]. In this study DCT was modeled in the form of 32 vector products. The entire DCT is a collection of 32 tasks, where each task is a vector product as shown in Figure 6. There are two kinds of tasks in the task graph, $T1$ and $T2$, whose structure is similar to the vector product, but whose bit-widths differ. A collection of eight tasks, forms a row of the 4x4 output matrix, as shown in the figure. The entire task graph consists of four such collections of tasks. Each task had three design points. These were carefully estimated using an estimation tool based on [18]. The functional units, area and latency for each is shown in Table 2. The result of the iterative refinement procedure for minimizing the latency of DCT for various FPGA resource bound, R_{max}, and reconfiguration overhead, C_T, values is shown in Tables 3 through 8. Run times for our temporal partitioning tool, in seconds, are shown for each iteration of the algorithm separately in the column T(s). The total run time in minutes for each experiment is shown in column T(m). All experiments have been run on an UltraSparc 1 machine running at 175 Mhz. In the first experiment, shown in Table 3, $R_{max} = 576$ CLBS and C_T is 30ns. The minimum number of partitions estimated by $MinAreaPartitions()$ is 8 and by $MaxAreaPartitions()$ is 11. We started our algorithms with $N = 9$, (Starting Partition Relaxation, $\alpha = 1$). We are able to reduce the latency of the circuit in steps by doing a binary division. Once the difference between

C_T	N	I	Bounds		Result		
			D_{max}(ns)	D_{min}(ns)	D_a	T(s)	T(m)
30ns $\alpha = 1$	9	1	25,710	1,065	9,650	37.40	
		2	7,226	1,065	7,060	77.32	
		3	4,145	1,065	Inf.	300	
		4	5,685	4,145	Inf.	300	
		5	6,455	5,685	Inf.	300	
		6	6,840	6,455	Inf.	300	
	10	1	7,060	1,095	6,500	278.8	
		2	4,077	1,095	Inf.	300	
		3	5,568	4,077	Inf.	300	
		4	6,314	5,568	Inf.	300	
		5	6,407	6,314	Inf.	300	
	11	1	6,500	1,125	Inf.	300	
	12	1	6,500	1,155	Inf.	300	56.55

Table 3. DCT, $R_{max} = 576, \delta = 200, \gamma = 1$

C_T	N	I	Bounds (without $N * C_T$)		Result		
			D_{max}(ns)	D_{min}(ns)	D_a	T(s)	T(m)
10ms $\alpha = 0$	8	1	25,440	795	Inf.	300	
	9	1	25,440	795	9,630	77.60	
		2	6,956	795	Inf.	300	
		3	9,266	6,956	9,100	78.95	
		4	8,111	6,956	8,100	185.73	
		5	7,533	6,956	7,380	281.93	
		6	7,244	6,956	Inf.	300	25.4

Table 4. DCT, $R_{max} = 576, \delta = 200, \gamma = 1$

the maximum and minimum latency bounds is less than $\delta = 200$, we stop. Then, we proceed by increasing N to 10 and repeat the latency refinement procedure. We sometimes need to have a timeout, either if the problem is infeasible or a solution is too difficult to find. This timeout is shown in the results as Inf.. Notice that, while we are tightening the latency constraint in each iteration of the solution, we are in effect making the solver progressively look at different parts of the design space. Since Ending Partition Relaxation, $\gamma = 1$, we stop our search at $N = 12$.

For the second experiment shown in Table 4, C_T is 10ms. For this experiment, we have not shown the value of reconfiguration overhead $N * C_T$ in the table. In this case we started with $\alpha = 0$, since the overhead of reconfiguration is very large, the least latency solution is most likely to be obtained in the minimum number of partitions. We start with 8 partitions, but no solution is possible. Then we relax the partition bound by 1, to 9 and continue the search for a solution. Notice that no relaxation of N was undertaken in this experiment, after a solution was achieved. This is because, the algorithm $Refine\_Partition\_Bound$ calculates the new minimum latency on relaxation, D_{min}, and finds that it is greater than the already achieved latency D_a, so it stops. In Table 5, we show the results on DCT with $R_{max} = 1024$. In this experiment the latency tolerance δ is 800. To show how varying the parameter δ affects the performance of the algorithm, we reduce δ to 100 and repeat the same experiment whose results are shown in Table 7. The number of iterations spent looking for a solution increases, thus increasing the runtime. But a better solution is achieved in iteration 6, at $N = 6$. The same results can be observed in Table 6 and Table 8, for the larger reconfiguration overhead of 10ms. Again, we see that reducing latency

			Bounds		Result		
C_T	N	I	D_{max}(ns)	D_{min}(ns)	D_a	T(s)	T(m)
30ns $\alpha = 1$	6	1	25,620	975	7,290	4.28	
		2	7,136	975	6,475	9.34	
		3	4,055	975	4,040	76.43	
		4	2,515	975	Inf.	300	
		5	3,285	2,515	Inf.	300	
	7	1	4,040	1,005	3,980	214.4	
		2	2,522	1,005	Inf.	300	
		3	3,281	2,522	Inf.	300	
	8	1	3,980	1,035	Inf.	300	30.07

Table 5. DCT, $R_{max} = 1024, \delta = 800, \gamma = 1$

			Bounds (without $N * C_T$)		Result		
C_T	N	I	D_{max}(ns)	D_{min}(ns)	D_a	T(s)	T(m)
10ms $\alpha = 0$	5	1	25,440	795	6,030	20.92	
		2	3,875	795	Inf.	300	
		3	5,222	3,875	Inf.	300	
		4	5,853	5,222	5,610	288.46	15.15

Table 6. DCT, $R_{max} = 1024, \delta = 800, \gamma = 1$

			Bounds		Result		
C_T	N	I	D_{max}(ns)	D_{min}(ns)	D_a	T(s)	T(m)
30ns $\alpha = 1$	6	1	25,260	975	7,290	4.28	
		2	7,136	975	6,475	9.34	
		3	4,055	975	4,040	76.43	
		4	2,515	975	Inf.	300	
		5	3,285	2,515	Inf.	300	
		6	3,670	3,285	3,640	104.04	
		7	3,477	3,285	Inf.	300	
		8	3,573	3,477	Inf.	300	
	7	1	3,640	1,005	Inf.	300	
	8	1	3,640	1,035	Inf.	300	33.23

Table 7. DCT, $R_{max} = 1024, \delta = 100, \gamma = 1$

			Bounds (without $N * C_T$)		Result		
C_T	N	I	D_{max}(ns)	D_{min}(ns)	D_a	T(s)	T(m)
10ms $\alpha = 0$	5	1	25,440	795	6,030	20.92	
		2	3,875	795	Inf.	300	
		3	5,222	3,875	Inf.	300	
		4	5,853	5,222	5,610	288.46	
		5	5,537	5,222	5,190	74.17	16.39

Table 8. DCT, $R_{max} = 1024, \delta = 100, \gamma = 1$

tolerance increases the run time but achieves better solutions. For all the experiments shown, we also experimented with obtaining optimal latency solutions as shown for the AR filter. However, in *none* of these experiments could the optimal solution process get even a single feasible solution in the same run time as the iterative solution process.

5 Conclusion

We have shown, that by using mathematical programming techniques we can model the task level temporal partitioning and design exploration problem incorporating multiple constraints of area, latency, and memory. We have also developed a framework in which these techniques can be used in a novel manner to solve constraint satisfaction problems for large specifications of real world examples such as the DCT. We are able to get near-optimal solutions in short run times with this iterative procedure. The effectiveness of the formulations and iterative procedure was demonstrated by the case study of the DCT.

The algorithms presented in this paper are integrated in the SPARCS (Synthesis and Partitioning for Adaptive Reconfigurable Computing Systems) [6, 7] design environment being developed at the University of Cincinnati. SPARCS is an integrated design system for automatically partitioning and synthesizing designs for reconfigurable boards with multiple field-programmable devices (FPGAs). The SPARCS system contains a temporal partitioning tool to temporally divide and schedule the tasks on the reconfigurable architecture, a spatial partitioning tool to map the tasks to individual FPGAs, and a high-level synthesis tool to synthesize efficient register-transfer level designs for each set of tasks destined to be down loaded on each FPGA. For more details go to http://www.ececs.uc.edu/∼ddel/projects/sparcs/sparcs.html.

References

[1] R. D. Hudson, D. I. Lehn and P. M. Athanas, “A Run-Time Reconfigurable Engine for Image Interpolation”, *IEEE Symposium on FPGAs for Custom Computing Machines, FCCM ’98.*

[2] M. J. Wirthlin and B. L. Hutchings, “Sequencing Run-Time Reconfigured Hardware with Software”, *ACM/SIGDA International Symposium on Field Programmable Gate Arrays, FPGA’96.*

[3] M. Vasiliko and D. Ait-Boudaoud, “Architectural Synthesis for Dynamically Reconfigurable Logic”, *International Workshop on Field-Programmable Logic and Applications, FPL’96.*

[4] M. B. Gokhale and J. M. Stone, “NAPA C:Compiling for Hybrid RISC/FPGA Architectures”, *IEEE Symposium on FPGAs for Custom Computing Machines, FCCM ’98.*

[5] J. Spillane and H. Owen, “Temporal Partitioning for Partially-Reconfigurable-Field-Programmable Gate”, *Reconfigurable Architectures Workshop in IPPS/SPDP’98.*

[6] I. Ouaiss, S. Govindarajan, V. Srinivasan, M. Kaul, and R. Vemuri, “An Integrated Partitioning and Synthesis System for Dynamically Reconfigurable Multi-FPGA Architectures”, *Reconfigurable Architectures Workshop in IPPS/SPDP’98.*

[7] S. Govindarajan, I. Ouaiss, M. Kaul, V. Srinivasan and R. Vemuri, “An Effective Design Approach for Dynamically Reconfigurable Architectures”, *IEEE Symposium on FPGAs for Custom Computing Machines, FCCM ’98.*

[8] M. Kaul and R. Vemuri, “Optimal Temporal Partitioning and Synthesis for Reconfigurable Architectures”, *Design and Test in Europe, DATE ’98.*

[9] C. H. Gebotys, “Optimal Synthesis of Multichip Architectures”, *IEEE ICCAD, p238-241, Nov. ’92.*

[10] C. H. Gebotys and M. I. Elmasry, “Optimal VLSI architectural synthesis”, *Kluwer Academic Publishers.*

[11] S. Trimberger, “Scheduling designs into a Time-Multiplexed FPGA”, *ACM/SIGDA International Symposium on Field Programmable Gate Arrays, FPGA’98.*

[12] S. Trimberger, “A Time-Multiplexed FPGA”, *IEEE Symposium on FPGAs for Custom Computing Machines, FCCM ’97.*

[13] R. Niemann and P. Marwedel, “An Algorithm for Hardware/Software Partitioning Using Mixed Integer Linear Programming”, *Proceedings of the European Design and Test Conference ED&TC ’96.*

[14] A. Kalavade, “System-Level Codesign of Mixed Hardware-Software Systems”, Ph.D. Dissertation, University of California, Berkeley, ’95.

[15] D. S. Rao and F. Kurdahi,“Hierarchical Design Space Exploration for a Class of Digital Systems”, *IEEE Transactions on VLSI, v 1, n 3, ’93.*

[16] M. Xu and F. Kurdahi, “Layout Driven High Level Synthesis for FPGA Based Architectures”, *Design and Test in Europe ’98.*

[17] J. Roy, N. Kumar and R. Vemuri, “DSS: A Distributed High-Level Synthesis System for $VHDL$ Specifications”, *IEEE Design and Test of Computers ’92* .

[18] R. Dutta, J. Roy, and R. Vemuri, “Distributed Design Space Exploration for High-Level Synthesis Systems”, *29th Design Automation Conference, June ’92.*

[19] G. D. Micheli, “Synthesis and Optimization of Digital Circuits”, *McGraw-Hill, ’94.*

[20] G.K. Wallace, “The JPEG Still Picture Compression Standard”, *ACM Communications, ’91.*

[21] Y. Hung, A. Parker, “High-Level Synthesis with Pin Constraints for Multiple-Chip Designs”, *29th Design Automation Conference, ’92.*

[22] WILDFORCE Reference Manual, *Document #1189 - Release Notes, Annapolis Micro Systems, Inc..*