Integrated Block-Processing and Design-Space Exploration in Temporal Partitioning for RTR Architectures *

Meenakshi Kaul
mkaul@ececs.uc.edu

Ranga Vemuri
Ranga.Vemuri@UC.EDU

Department of ECECS, University of Cincinnati, Cincinnati, OH 45221–0030

Abstract. *We present an automated temporal partitioning and design space exploration methodology that temporally partitions behavior specifications. We propose block-processing in the temporal partitioning framework for reducing the reconfiguration overhead for partitioned designs. Block-processing is a technique used traditionally in the area of parallel compilers, for increasing the computation speed by processing several inputs simultaneously. Block-processing technique has been integrated with task-level design space exploration to achieve designs that justify temporal partitioning of systems. An ILP-based methodology has been proposed to solve this problem. We present experimental results for the Discrete Cosine Transform (DCT).*

1 Introduction

The reconfiguration capability of SRAM-based FPGAs can be utilized to fit a large application onto an FPGA by partitioning the application *over time* into multiple segments. This division into temporal segments is called *temporal partitioning*. Such temporally partitioned applications are also called Run-Time Reconfigured (RTR) systems.

To overcome the effects of high reconfiguration overhead for existing reconfigurable hardware, we demonstrated in [8] how loop fission techniques can be used as a post-processing step after temporally partitioning a design to increase the throughput. In the current work, we extend our temporal partitioning technique to incorporate block-processing and design space exploration techniques and demonstrate how this integrated processing can be used to search for optimized temporally partitioned designs.

We assume the input specification to be a task graph [4], where each task consists of a set of operations. 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* 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 optimal design point for a task in the context of optimizing the overall throughput of the design will depend on the architectural constraints and the dependency constraints among the tasks. 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.

Block-Processing in Temporally Partitioned designs : In many application domains eg., Digital Signal Processing, computations are defined on very long streams of input data. In order to decrease overall execution time multiple inputs from the input data stream can be processed simultaneously. This approach known as *block-processing*, is used to increase the throughput of a system through the use of parallelism and pipelining in the area of parallel compilers [12] and VLSI processors [13]. We can

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

apply the concept of block-processing to a single processor reconfigurable system to speedup the processing time.

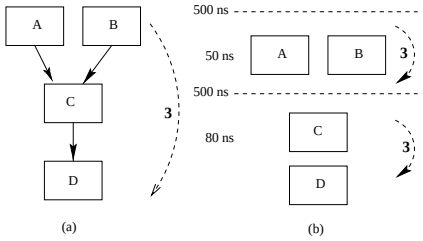

Fig. 1. Temporally partitioned design example

Figure 1 illustrates the use of block-processing to speed up computation in a temporally partitioned design. The task graph consists of 4 tasks A, B, C, D. It is partitioned into two temporal partitions as shown in Figure 1(b). The latency of temporal partition 1 is 50ns and of partition 2 is 80 ns. The reconfiguration time is 500 ns. The latency of the design is 50+500 +80+500 = 1130 ns. A single iteration of the task graph executes in 1130 ns. Now three iterations of this temporally partitioned design will take 3x1130 = 3390 ns. However if we perform block-processing by sequencing all 3 computations on each temporal partition, the time taken for the execution is (50+50+50)+500+(80+80+80)+500 = 1390 ns. Thus block-processing amortizes the reconfiguration overhead over the 3 inputs. Block-processing is possible only for applications that process a large stream of inputs. We represent such applications by a task graph having an implicit outer loop as shown in Figure 1(a). Note that block-processing is possible if there are no dependencies among the computations for different inputs. In compiler terminology this means there should be no loop-carried dependencies due to the implicit outer loop among different iterations of this loop. In this paper, we deal with applications for which no dependencies among computations is present. Most DSP applications fall in this category.

Integrating Design-Space Exploration and Block-Processing in Temporal Partitioning : For FPGA based architectural synthesis, the constraints of area in terms of CLBs (Configurable Logic Blocks)/FGs (Function Generators) and memory are to be met by 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.

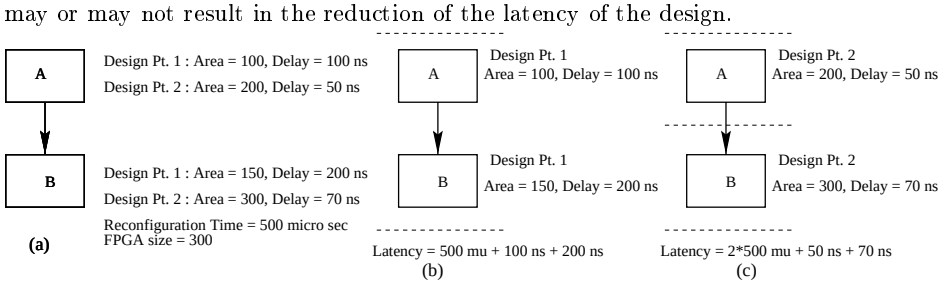

Fig. 2. Design space exploration

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, it is *not necessary* that the minimum latency design is the best solution. We illustrate this idea with an example. In the

Figure 2(a) a task graph is shown. Each task has two different design points on which it can be mapped. Two different solutions (b) and (c) are possible. If minimum latency solution is required then solution (b) will be chosen over solution (c) because the latency of (b) is 500.3 μ sec and latency of (c) is 1000.12 μ sec. Now, if we use (b) and (c) in the block-processing framework to process 5000 computations on each temporal partition. Then the execution time for solution (b) is 2000 μ seconds and for solution (c) is 1600 μ seconds. Therefore if we can integrate the knowledge about block-processing while design space exploration is being done then it is possible to choose more appropriate solutions.

The price paid for block-processing is the higher memory requirements for the reconfigured design. We call the number of data samples or inputs to be processed in each temporal partition to be the the *block-processing factor*, k. This is given by the user and is the minimum number of input data computations that this design will execute for typical runs of the application. The amount of block-processing is limited by the amount of memory available to store the intermediate results.

Currently many designers perform temporal partitioning manually [1] or the designer specifies the partitioning points to the tool [3]. Existing automated temporal partitioning techniques, extend existing scheduling techniques of high-level synthesis [2, 9] and focus on minimizing the number of partitions of the design. In [7], we presented a mathematical model for combined temporal partitioning and synthesis. In that approach, synthesis cost exploration is performed at an operation level in the task graph, and the number of alternative solutions explored becomes very large. It can be used to synthesize small-scale behavior specifications. Our current technique can also simultaneously handle multiple design constraints, eg., FPGA resources, on-board memories, and perform design exploration for large specifications.

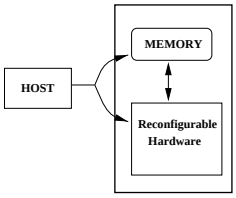

Fig. 3. RTR System Architecture

2 System Architecture and Design Flow

In Figure 3, the hardware architecture on which the Run-Time reconfigured design is to be mapped is shown. It consists of a reconfigurable hardware communicating with an external memory. Each temporal partition is mapped to the reconfigurable hardware, and the data flowing between two temporal partitions is mapped to the memory. A Host interacts with both the reconfigurable hardware and the memory and is used to load new configurations. In Figure 4, we present the design flow for building a Run-Time Reconfigured (RTR) design. The input specification, is a behavior level design description of the application to be implemented on the reconfigurable hardware. The input specification consists of acyclic data flow task graph, with an outer implicit loop. The implicit loop signifies the successive items of input data that will be executed on this task graph.

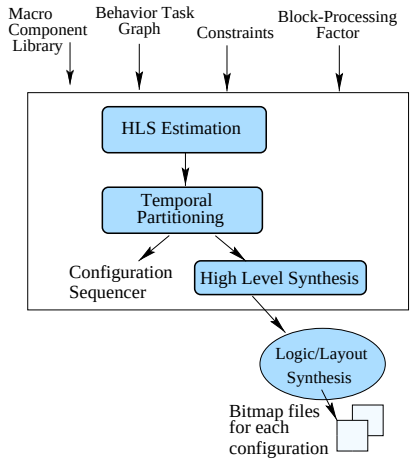

Fig. 4. Design Flow

Task Estimation: First, the behavior level estimation engine, part of a High Level

synthesis tool, DSS [10], generates multiple design points for each task separately based on the architecture and user constraints. The architecture constraints are the resources available on the reconfigurable hardware, the user constraints are the maximum clock-width for the design. The HLS tool makes use of a component library characterized for the particular reconfigurable device, to estimate the resource and delay.

Temporal Partitioning: Next, the temporal partitioning tool divides the task graph into multiple temporal segments, while mapping each task to its appropriate design point. We discuss the ILP formulation used to solve the multi-constraint temporal partitioning problem later in detail.

High Level Synthesis: A high level synthesis system is used to generate the RTL design for each temporal segment.

Logic/Layout Synthesis: We use commercial tools, for logic synthesis (Synplify tools from Synplicity) and layout synthesis (Xilinx M1 tools) to convert the RTL description of each configuration into bitmap files.

3 Temporal Partitioning

The inputs to our Temporal Partitioning system are - (1) Behavior specifications (2) Target Architecture Parameters (3) Block-processing Factor

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}	*memory size of the RTR architecture.*
C_T	*reconfiguration time of the reconfigurable processor.*
k	*the block-processing factor for the design.*

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. 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. k, the block-processing factor is the lower bound on the number of computations that this design will usually perform.

3.1 Preprocessing

Design Point Generation: Each task in the task graph is processed by a design space exploration and estimation tool which is part of a high level synthesis system. The high level synthesis tool generates a set of *design points* for each task. Each design point has an associated *module set* [11]. 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.
$N^l_{min} = \sum_{t \in T} R(m)/R_{max}, \quad m \in M_t,$ m is the minimum area module set in M_t
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 define, the minimum number of partitions, N^u_{min}, that need to be explored if *maximum* area design point for each task is mapped by the partitioner, to be -
$N^u_{min} = \sum_{t \in T} R(m)/R_{max}, \quad m \in M_t,$ m is the maximum area module set in M_t

3.1.1 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 an iterative procedure, shown in Figure 5. We calculate the bounds on the number of partitions, N^l_{min} and N^u_{min}, as described earlier. We start the search at N^l_{min} and obtain an optimal solution, by forming and solving an ILP model of the temporal partitioning problem. The details of the model are described in Section 3.1.2. For the first ILP model there is no upper bound on constraint on the execution time of the design. The result of solving this model is a temporal partitioning solution for N partitions and the execution time D_a of the solution. We now relax N by 1, form and solve the ILP model again. This time however, since we are looking for a better solution than the one we have already achieved, D_a is the execution time constraint for the new ILP model. 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 *Partition Relaxation*, which defines the number of partitions beyond N^u_{min} that must be explored while searching for better solutions.

```
Algorithm Refine_Partition_Bound()
begin
  N^u_min ← MaxAreaPartitions()
  N^l_min ← MinAreaPartitions()
  N ← N^l_min /* starting partition number */
  D_a ← Solve_ILP_Model()
  while D_a = 0 /* Partition bound was infeasible */
    N ← N + 1 /* next partition number */
    D_a ← Solve_ILP_Model()
  end while
  while N < N^u_min + γ and not TimeExpired()
    N ← N + 1 /* Relax N */
    D'_a ← Solve_ILP_Model()
    if D'_a ≠ 0 /* solution is feasible */
      D_a ← D'_a
    end if
  end while
  return(D_a) /* return with the last known best solution */
end Algorithm Refine_Partition_Bound
```

Fig. 5. Partition Refinement Procedure

3.1.2 ILP formulation for Design Space Exploration

We build the temporal partitioning model for the given tasks and their design points and the values of N and k. 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 -

N	*bound on the number of partitions.*
D_a	*constraint on the execution time of the design.*
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.

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

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

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

$d_p =$ execution time 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.

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

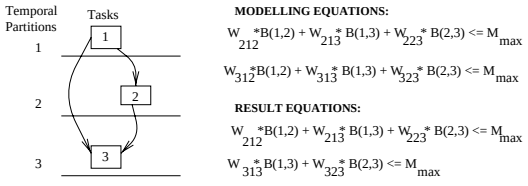

Fig. 6. Memory Constraints

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.

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

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 6, 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 -

$$\forall p,\ \ 1 \leq p \leq N,\ \ \forall t_2 \in T,\ \ \forall t_1 \rightarrow t_2,\ \ \ : w_{pt_1t_2} \geq \sum_{1 \leq p_1 < p} y_{t_1p_1} * \sum_{p \leq p_2 \leq N} y_{t_2p_2} \quad (4)$$

$$\forall p,\ \ 1 \leq p \leq N,\ \ \forall t_2 \in T,\ \ \forall t_1 \rightarrow t_2,\ \ \ : w_{pt_1t_2} \geq y_{t_1p} * \sum_{p+1 \leq p_2 \leq N} y_{t_2p_2} \quad (5)$$

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 [7] 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.

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

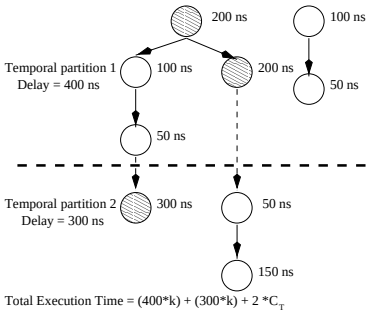

Fig. 7. Execution Time Estimation

Execution Time Constraint: The execution time of a partition will be the maximum execution time among all the paths of the task graph mapped to that partition. In Figure 7, we show how the execution time 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. If the block-processing factor is k, then the execution time of the partition is the latency multiplied to the block-processing factor. Formally the execution time of a temporal partition is given as -

$$\forall p,\ \ 1 \leq p \leq N,\ \ \forall (t_i \overset{p}{\rightarrow} t_j) \in P_{l \overset{p}{\rightarrow} r}\ \ \ : \sum_{m \in M_t} \sum_{t \in t_i \overset{p}{\rightarrow} t_j} (y_{tpm} * D(m) * k) \leq d_p \quad (7)$$

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 -

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

Now the execution time constraint on the overall design can be stated in terms of equations (7) and (8) as -

$$\eta * C_T + \sum_{p=1}^{N} d_p \leq D_a \quad (9)$$

As discussed earlier, this constraint is used to search for a better solution as different partition bounds are being explored in Algorithm *Refine_Partition_Bound* in Figure 5.

Optimality Goal: The most optimal solution will be the design with the least execution time.

$$Minimize\ \ \ : \eta * C_T + \sum_{p=1}^{N} d_p \quad (10)$$

The solution of this ILP model gives us the optimum temporal partitioning for the give partition bound N, the block-processing factor k, and the set of design points for the tasks. If the amount of intermediate memory required to process k computations exceeds the memory constraint M_{max} of the architecture then the user needs to reduce

k and temporally partition the design again.

4 Experimental Results

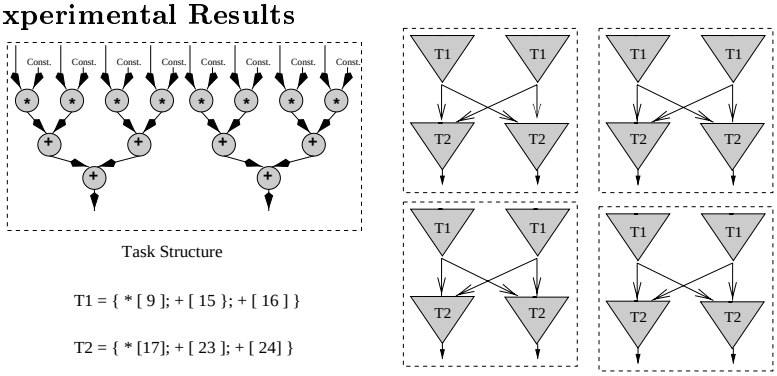

Fig. 8. Task graph for DCT

We performed temporal partitioning on the Discrete Cosine Transform (DCT) [16] which is the most computationally intensive part of the JPEG [17] algorithm. In this study, the DCT is a collection of 16 tasks, the structure of a task is shown in Figure 8. There are two kinds of task in the task graph, $T1$ and $T2$, whose structure is similar but whose operations have different bit widths. We obtained all the design points for each kind of tasks, by using estimation tools on the individual tasks. The functional units, area and latency costs for each is shown in Table 1.

		Characteristics					
Task	Desgn. Pt.	Area (CLBs)	Latency (ns)	$*_9$	$+_{16}$	$*_{16}$	$+_{24}$
T1	1	336	375	8	4		
	2	286	500	6	2		
	3	220	625	4	2		
	4	194	750	2	2		
	5	174	875	1	2		
T2	1	396	420			8	4
	2	356	560			6	2
	3	292	700			4	2
	4	276	840			2	2

Table 1. Design Points for DCT tasks

In Tables 2 - 4 we present the results of our temporal partitioning tool. In all the tables, R_{max} is the resource constraint of the FPGA, C_T the reconfiguration time, k the block-processing factor, and N the number of partitions onto which the design is partitioned. The latency of the final design (with the reconfiguration overhead), is shown in the column Latency. The execution time of the design for the k blocks of data is given in the column Design Execution Time. Partitioner Run Time in seconds is the time taken by our temporal partitioning tool to execute. All experiments were run using an ILP solver called CPLEX on an UltraSparc Machine running at 175 MHz. In Table 2, we present the result of temporal partitioning and design space exploration of the DCT for with and without block-processing factors. In Exp. 1, for a block-processing factor of 3,000, our temporal partitioning tool explores 3 temporal partitions for the design and results in a latency of 60,795 ns. In Exp. 2, with a block-processing factor of 1 (i.e., no computations are being sequenced), the tool gives a minimum latency design of 31,590 ns and uses just one temporal partition. This results in a statically configured design. Even though, the latency of the statically configured design in Exp. 2 is less than that of Exp. 1, this is not the best possible solution. This is because, if multiple computations are computed on both the static and RTR design, the RTR

Exp.	R_{max} (CLBs)	C_T (μs)	k	N	Latency (ns)	Design Execution Time	Partitioner Run Time (s)
1	4,000	30	3,000	1	31,590	4,800 μs	1
				2	60,795	2,445 μs	1
				3			Infeasible
2	4,000	30	1	1	31,590		1
3	2,304	30	3,000	2	61,590	4,830 μs	11
				3	91,215	3,735 μs	22
				4			Infeasible
4	2,304	30	1	2	61,590		57

Table 2. Results for combined design-space exploration and block-processing

design will outperform the static design. For executing 3,000 computations, the RTR design will take 2,445 μ sec, while the static design will take 4,800 μ sec. This is a 49% improvement of the RTR design over the static design. Exp. 3 and 4 were performed for different FPGA size of 2,304 CLBs. In Exp. 3, again with a block-processing factor of 3000, the optimal design takes 3 temporal partitions with the latency of the design being 91,215 ns. For Exp. 4, the optimal latency of the design is 61,590 ns. Again, the actual execution time of the design when the block-processing factor is considered while exploring the design space is superior. In all experiments in Table 2 the reconfiguration time considered is similar to the Xilinx 6000 series FPGAs.

The experiments in Table 2 illustrate that combining block-processing and design space exploration gives better temporal partitioning solutions. If the block-processing factor is equal to 1, then the temporal partitioning tool will tend to pick the design with minimum number of temporal partitions. If a relevant block-processing factor is given the tool will search for a design with more temporal partitions, because block-processing will amortize the effects of reconfiguration overhead. So, since we understand that the block-processing is necessary for good performance of a temporally partitioned design, we must integrate this idea early in the design process, while partitioning and design point selection is being performed.

Similar results will hold if the reconfiguration overheads are varied. In Table 3, we show results for different reconfiguration overheads. In Exp. 5 and 6, the reconfiguration overhead is in nano-seconds (similar to the reconfiguration overheads of context-switching FPGAs like the Time Multiplexed FPGA [14]). In Exp. 7 and 8, the reconfiguration overhead is in milli-seconds (similar to commercially available reconfigurable hardware [15]). As the reconfiguration overhead decreases we observe that for small values of k, the exploration process chooses more temporal partitions. However, for the reconfiguration overheads in milli-seconds even for values of k as large as 3,000 the temporal partitioner chooses designs with minimum temporal partitions.

Exp.	R_{max} (CLBs)	C_T	k	N	Latency (ns)	Design Execution Time	Partitioner Run Time (s)
5	2,304	30 ns	300	2	1,650	477.06 μs	17
				3	1,305	364.59 μs	19
6	2,304	30 ns	50	2	1,650	79.56 μs	25
				3	1,305	60.84 μs	7
7	2,304	3 ms	3,000	2	6,001,590	10,770 μs	80
8	2,304	3 ms	30,000	2	6,001,590	53,700 μs	29
				3	9001,215	45,450 μs	36

Table 3. Results for different reconfiguration overheads

In Table 4, we illustrate how design space exploration is beneficial. For same values of the block-processing factor k, we perform experiment with and without design space exploration. In Exp. 9, temporal partitioning is performed with only one design point for each task, the minimal area design point. In Exp. 10, all the design points are used. Again we observe that the tool chooses the most appropriate design points for the given constraints, when multiple design points are given to it, and results in a 27%

improvement of the design in Exp. 10. Therefore design space exploration must be integrated in the temporal partitioning process, rather than choosing the design point before temporal partitioning is performed.

Exp.	R_{max} (CLBs)	C_T (μs)	k	N	Latency (ns)	Design Execution Time	Partitioner Run Time (s)
9	2,304	30	3,000	2	31,715	5,145.6 μs	1
10	2,304	30	3,000	2	61,590	4,830 μs	22
				3	91,215	3,735 μs	204

Table 4. Results for design-space exploration

5 Conclusion

The algorithms presented in this paper are integrated in the SPARCS (Synthesis and Partitioning for Adaptive Reconfigurable Computing Systems) [5, 6] 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, a spatial partitioning tool, and a high-level synthesis tool. 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 1998, pp. 88-95.*
2. M. Vasiliko and D. Ait-Boudaoud, "Architectural Synthesis for Dynamically Reconfigurable Logic",*International Workshop on Field-Programmable Logic and Applications, FPL 1996, pp. 290-296.*
3. M. B. Gokhale and J. M. Stone, "NAPA C:Compiling for Hybrid RISC/FPGA Architectures", *IEEE Symposium on FPGAs for Custom Computing Machines, FCCM 1998, pp. 126-135.*
4. I. Ouaiss, S. Govindarajan, V. Srinivasan, M. Kaul, and R. Vemuri, "A Unified Specification Model of Concurrency and Coordination for Synthesis from VHDL", *International Conference on Information Systems, Analysis and Synthesis, ISAS 1998, pp. 771-778.*
5. 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 12th International Parallel Processing Symposium and 9th Symposium on Parallel and Distributed Processing, IPPS/SPDP 1998, pp. 37-42.*
6. 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 1998, pp.312-313.*
7. M. Kaul and R. Vemuri, "Optimal Temporal Partitioning and Synthesis for Reconfigurable Architectures", *Design and Test in Europe, DATE 1998, pp. 389-396.*
8. M. Kaul, R. Vemuri, S. Govindarajan and I. Ouaiss, "An Automated Temporal Partitioning Tool for a class of DSP applications", *Workshop on Reconfigurable Computing in International Conference on Parallel Architectures and Compilation Techniques, PACT 1998, pp. 22-27.*
9. S. Trimberger, "Scheduling designs into a Time-Multiplexed FPGA", *ACM/SIGDA International Symposium on Field Programmable Gate Arrays, FPGA 1998, pp. 153-160.*
10. J. Roy, N. Kumar and R. Vemuri, "DSS: A Distributed High-Level Synthesis System for *VHDL* Specifications", *IEEE Design and Test of Computers, v9, n2, June 1992, pp. 18-32.*
11. R. Dutta et. al, "Distributed Design Space Exploration for High-Level Synthesis Systems", *29th Design Automation Conference, DAC 1992, pp. 644-650.*
12. M. Wolf, *High Performance Compilers for Parallel Computing*, Addison-Wesley Publishers, 1996.
13. S. Y. Kung, *VLSI Array Processors*, Prentice Hall 1988.
14. S. Trimberger, "A Time-Multiplexed FPGA", *IEEE Symposium on FPGAs for Custom Computing Machines, FCCM 1997, pp. 22-28.*
15. WILDFORCE Reference Manual, *Document #1189 - Release Notes, Annapolis Micro Systems, Inc..*
16. N. Narasimhan, V. Srinivasan, M. Vootukuru, J. Walrath, S. Govindarajan and R. Vemuri, "Rapid Prototyping of Reconfigurable Coprocessors", *International Conference on Application-Specific Systems, Architectures and Processors, 1996.*
17. G.K. Wallace, "The JPEG Still Picture Compression Standard", *ACM Communications, 1991.*

This article was processed using the LaTeX macro package with LLNCS style