An Automated Temporal Partitioning and Loop Fission approach for FPGA based reconfigurable synthesis of DSP applications *

Meenakshi Kaul, Ranga Vemuri, Sriram Govindarajan and *Iyad Ouaiss*
{mkaul,ranga,sriram,iouaiss}@ececs.uc.edu
Digital Design Environments Laboratory, University of Cincinnati, Cincinnati, OH 45221-0030

Abstract

We present an automated temporal partitioning and loop transformation approach for developing dynamically reconfigurable designs starting from behavior level specifications. An Integer Linear Programming (ILP) model is formulated to achieve near-optimal latency designs. We, also present a loop restructuring method to achieve maximum throughput for a class of DSP applications. This restructuring transformation is performed on the temporally partitioned behavior and results in near-optimization of throughput. We discuss efficient memory mapping and address generation techniques for the synthesis of reconfigurable designs. A Case study on the Joint Photographic Experts Group (JPEG) image compression algorithm demonstrates the effectiveness of our approach.

1 Introduction

The reconfiguration capability of the SRAM FPGAs can be utilized to fit a large application onto the FPGA by partitioning the application *over time* into multiple segments. The division of an application into temporal segments that are configured one after the other on the FPGA, is called temporal partitioning. The first temporal partition receives input data, performs computations and stores the intermediate result into the on-board memory. The device is then reconfigured for the next segment, which computes results based on the intermediate data, from the previous partition. This process is repeated until all the partitions are executed. Such temporally partitioned designs are called Run-Time Reconfigured (RTR) systems.

In this paper, we concentrate on behavior level design descriptions to be temporally partitioned. Since the reconfiguration overhead for currently available hardware is usually orders of magnitude larger than the latency of the design, we need to concentrate on throughput maximization techniques to minimize the effects of the reconfiguration overhead. For throughput maximization, we present an automated technique for DSP style applications, that automatically sequences multiple computations in each temporal partition to reduce the reconfiguration overhead. We use a loop transformation approach called *loop fission*. To our knowledge, no existing tools perform automated loop transformation to reduce the reconfiguration overhead in the context of reconfigurable processors. An experiment on the JPEG [17] algorithm demonstrates the effectiveness of our approach.

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

DAC 99, New Orleans, Louisiana

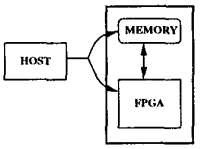

Figure 1: **RTR System Architecture**

Currently many designers perform temporal partitioning manually [1, 2] or the designer needs to specify the partitioning points of the application [3]. Others, have addressed temporal partitioning heuristically, by extending existing scheduling and clustering techniques of high-level synthesis [4, 5, 6, 8]. In an earlier work [7] we presented a mathematical model for simultaneous temporal partitioning and synthesis on operation level data-flow graphs. In that approach, synthesis cost exploration was performed at an operation level in the task graph. Due to the size of the formulation, small behavior specifications can be handled. In the current work the temporal partitioning tool performs exploration at the task level. It can handle large designs and can simultaneously handle multiple design constraints eg., CLBs, on-board memories while obtaining minimum latency (delay) designs, that cannot be handled by current techniques.

We present the system architecture and design flow in Section 2, the temporal partitioning model in Section 2.1 and the loop fission technique in Section 2.2. Extensions to support loop fission are discussed in Section 3, experimental study in Section 4 and conclusions in Section 5.

2 System Architecture and Design Flow

In Figure 1, the hardware architecture on which the Run-Time reconfigured design is to be mapped is shown. It consists of an FPGA communicating with an external memory. Each temporal partition is mapped to the FPGA, and the data flowing between two temporal partitions is mapped to the memory. A Host interacts with both the FPGA and the memory and is used to load new configurations and store the intermediate memory on the host, if needed. In Figure 2, 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 is shown in Figure 3. It consists of data flow graph, with an outer implicit loop.

Task Estimation: First, the behavior level estimation engine, part of a High Level synthesis tool, DSS [13], estimates the resources and execution delay or latency for each task

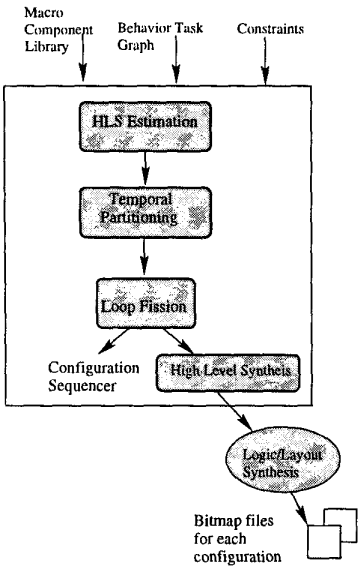

Figure 2: **Design Flow**

separately based on the architecture and user constraints. The architecture constraints are the resources available on the FPGA, 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. To bridge the gap between behavior and the final layout on the FPGA, floor planning based layout estimation techniques [10, 11] are incorporated in the estimation engine.

Temporal Partitioning: Next, the temporal partitioning tool divides the task graph into multiple temporal segments, that run one after the other on the reconfigurable device. We discuss the ILP formulation used to solve the multi-constraint temporal partitioning problem later in detail.

Loop Fission: The output from the temporal partitioning engine is the optimal latency design feasible. However, since the reconfiguration overheads of currently available reconfigurable systems is usually very large, an RTR design will not in general give better performance than its static counterpart. This analysis tool is designed for a class of DSP applications, where the task graph of behavior specification is executed inside an iterative loop. The same computation needs to be performed a large number of times depending on the input data, which is known only at runtime. A loop restructuring (loop fission [14]) technique will be used to maximize throughput for such applications, while minimizing the reconfiguration overhead. The output of this step is mapping of tasks to the temporal partitions and the design transformation needed to run more than one computation on the same temporal partition. It also generates a software code to sequence the configurations from the host.

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 for the FPGA.

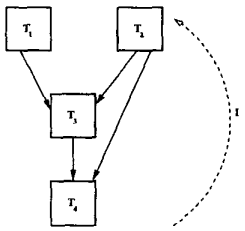

Figure 3: **Behavior Task Graph with implicit loop**

2.1 ILP formulation for Temporal Partitioning

Mathematical models of other partitioning and co-design problems have been addressed by researchers. The problem of spatial partitioning and synthesis was formulated as an ILP in [15]. Hardware software partitioning of co-design systems is addressed in [16]. The inputs to the Temporal Partitioning tool are :

(1) Behavior specifications (2) Task synthesis costs (3) Target Architecture Parameters

The behavior specifications are in the form of a *Task Graph*, as shown in Figure 3. For each task, the HLS tool generated the synthesis costs in terms of the FPGA resources and execution delay of the task, as explained earlier.

The architecture constraints are stated in formal notation as

R_{max}	*resource capacity of the FPGA.*
M_{max}	*temporary on-board memory size.*
C_T	*reconfiguration time for the FPGA.*

Typically resource capacity R_{max}, is the combinational logic blocks/function generators on the FPGAs of the reconfigurable device. M_{max}, is the on-board memory size. The reconfiguration time for reconfiguring the FPGA is C_T.

Preprocessing Step: To build the ILP model, we need to determine the number of partitions for which a solution is to be obtained. We start by getting a lower bound on the number of partitions for the particular problem. We sum the resources for all tasks. This value divided by the FPGA resource will be the minimum number of partitions required to obtain a solution.

Model Generation and Solution: We build the temporal partitioning model for the given inputs and linearize the non-linear constraints. We then solve it using a linear programming solver. If the solution is infeasible, we relax the partition bound N by 1, and rebuild and solve the model till we get a solution. The solution obtained is optimal for the given task graph.

The following notation will be used in generating the ILP model constraints:

N	*bound on the number of partitions, 1 ... N is the order of execution of the partitions.*
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.*
$R(t)$	*resources for task t, obtained from HLS tool.*
$D(t)$	*delay of task t, obtained from HLS tool.*

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* $t_j \in T$ *in the task graph.*
$P_{l \xrightarrow{p} r}$	$\{ t_i \xrightarrow{p} t_j \mid (t_i \in T_r) \wedge (t_j \in T_l)\}$, *(set of paths from root tasks to leaf tasks).*

Variables and Constraints

Variable y_{tp}, models partitioning. Data transfer across a temporal partition required due to two dependent tasks, is modeled by $w_{pt_1t_2}$. d_p, models the execution time of the temporal partitions.

$$y_{tp} = \begin{cases} 1 & \text{if task } t \in T \text{ is placed in partition p,} \\ & 1 \leq p \leq N \\ 0 & \text{otherwise} \end{cases}$$

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

$$d_p = \text{ delay of partition } p.$$

Variables y_{tp}, $w_{pt_1t_2}$ are 0-1 variables, d_p can be integer or real depending on whether the delay values are integer or real.

Uniqueness Constraint: Each task should be placed in exactly one partition among the N temporal partitions.

$$\forall t \in T \quad : \sum_{p=1}^{N} y_{tp} = 1 \tag{1}$$

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.

$$\forall t_2, \ \forall t_1 \rightarrow t_2, \ \forall p_2, \ 1 \leq p_2 \leq N-1 \quad :$$

$$\sum_{p_2 < p_1 \leq N} y_{t_1p_1} + y_{t_2p_2} \leq 1 \tag{2}$$

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.

$$\forall p, \ 1 \leq p \leq N \quad :$$

$$\sum_{t \in T} \sum_{p \leq p_2 \leq N} y_{tp_2} * B(env, t) + \sum_{t \in T} \sum_{1 \leq p_3 \leq p} y_{tp_3} * B(t, env) +$$

$$\sum_{t_2 \in T} \sum_{t_1 \rightarrow t_2} (w_{pt_1t_2} * B(t_1, t_2)) \leq M_{max} \tag{3}$$

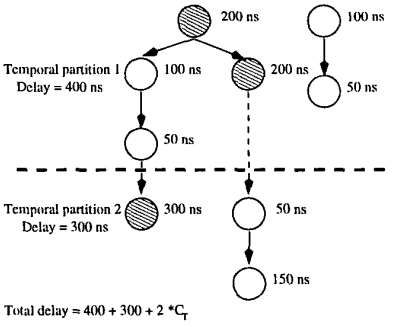

Figure 4: **Delay Estimation**

$w_{pt_1t_2}$ are 0-1 non-linear terms constrained as -

$$\forall p, \ 1 \leq p \leq N, \ \forall t_2 \in T, \ \forall t_1 \rightarrow t_2, \quad :$$

$$w_{pt_1t_2} \geq \sum_{1 \leq p_1 < p} y_{t_1p_1} * \sum_{p \leq p_2 \leq N} y_{t_2p_2} \tag{4}$$

$$w_{pt_1t_2} \geq y_{t_1p} * \sum_{p+1 \leq p_2 \leq N} y_{t_2p_2} \tag{5}$$

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 [7] to solve the combined temporal partitioning and synthesis problem.

Resource Constraint: The sum of resource 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 (CLB) etc. Similar equations can be added if multiple resource types exist in the FPGA.

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

Optimality Goal: The delay of design execution on a partition will be the maximum delay among all the paths of the task graph mapped to that partition. We see in Figure 4, how the delay for a partition is determined. The final mapping of tasks to partitions, with the delay value for each task, is shown. In partition 1, 3 paths are mapped. The delay of this partition is the greatest delay along a path mapped to the partition, i.e., maximum among 350ns, 400ns, 150ns. The maximum delay in partition 2 is 300ns. Formally the delay of design execution on a temporal partition is given as -

$$\forall p, \ 1 \leq p \leq N, \ \forall (t_i \xrightarrow{p} t_j) \in P_{l \xrightarrow{p} r} \quad :$$

$$\sum_{t \in t_i \xrightarrow{p} t_j} (y_{tp} * D(t)) \leq d_p \tag{7}$$

Now the most optimal solution will be the design with the least delay of execution. The minimization goal is stated as -

$$Minimize \quad N * C_T + \sum_{p=1}^{N} d_p \tag{8}$$

The result obtained by solving this ILP model, will produce a minimum latency design for the given task graph.

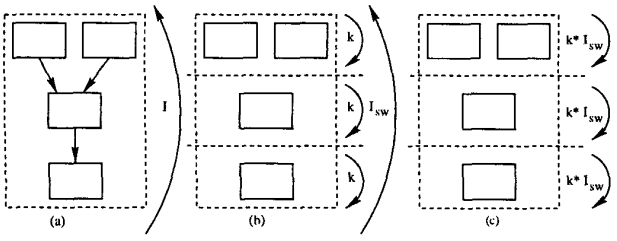

Figure 5: **Strategy for processing Multiple Computations**

2.2 Loop Fission

The reconfiguration time for a reconfigurable FPGA varies from nano-seconds, as in the *Time Multiplexed FPGA* [9], to milli-seconds as in the commercially available FPGA boards eg., *WildForce* [18]. If the overhead of reconfiguration is much larger than time for executing the application, the performance of a statically configured design may outperform a run-time reconfigured design. The decrease in the latency of the design due to increased availability of resources due to reconfiguration, is dwarfed by the reconfiguration overhead. In this section, we concentrate on FPGA's whose reconfiguration overhead is in milliseconds.

We noticed, that for a host of DSP style applications eg., Image processing, Template Matching, Encryption algorithms there is an implicit outer loop for the specification which depends on the number of inputs to be processed. For eg., the JPEG image compression algorithm compresses image files of various sizes. Such computation can be seen as a simple loop enclosing the task graph, whose loop count can be known only at run-time.

Multiple Computations on a static design: For a static design (without run-time reconfiguration), all the inputs can be processed sequentially on the same static circuit.

Multiple Computations on a RTR design: For a run-time reconfigured design this sequencing of inputs will not work. This is because, when the first input is processed by a temporal partition the intermediate result will be written out and the next temporal partition loaded. So unless explicit provision has been made in the design to process more than one set of inputs in a temporal partition, more than one set of inputs cannot be processed in the same temporal partition. This implies that, if k inputs have to be processed for correct output to be produced all the temporal partitions have to be loaded afresh for processing each input and so the overhead of reconfiguration becomes $k * N * C_T$.

To overcome this problem, and reduce the effect of reconfiguration, the RTR circuit has to be synthesized such that it processes multiple inputs in the first temporal partition sequentially, and write out all the intermediate data. The subsequent partitions must be capable again of reading multiple inputs and processing them sequentially. Then if we assume that memory size is not a limitation, the reconfiguration overhead for processing k inputs is still $N * C_T$. The frequency of reconfiguration has been reduced, thus reducing the reconfiguration overhead while the throughput of the design increases.

However, if only limited memory is available to store the inputs or intermediate data, not all inputs can be processed together in each partition. We therefore also perform an analysis of the number of computations k, that can be performed in the amount of memory available. We use the following terms for the ensuing discussion:

I	*total number of computations to be performed.*
m^i_{temp}	*size of intermediate memory required in each partition $1 \leq i \leq N$, for each computation.*
k	*total number of computations that can be performed in each temporal partition*
D_{mem}	*delay in communicating 1 memory element between the host & the memory of the FPGA.*

The following equation gives the total number of inputs that can be processed, by each temporal partition due to memory constraints.

$$k = \lfloor M_{max} / \max(m^i_{temp}) \quad \forall i,\ 1 \leq i \leq N \rfloor \qquad (9)$$

Software sequencing of RTR design: As discussed above, k computations are performed in one run of the RTR design. To perform all the I computations, we need to rerun the RTR design with the next set of k computations and so on, till all the inputs have been processed. This sequencing is done by a software loop which is executed in the host, and is used to load the RTR design as many times as needed. The loop count of this software loop is $I_{sw} = \lceil I/k \rceil$

In Figure 5, we show symbolically the loop fission being done in this step. In 5(a) we show the original task graph, that has to be executed I number of times as depicted by the outer loop. We show two software sequencing techniques, when the task graph has been split into three temporal partitions. In 5(b) in each temporal partition, k computations will be performed, and the next temporal partition will be executed. Once, all three temporal partitions are executed on the first set, k, of computations, The output will be sent to the host and the next k computations is loaded. We need to run the reconfiguration sequence on the device all over again after every k computations. The reconfiguration overhead associated with this strategy is $N * C_T * I_{sw}$. This technique of throughput maximization is referred to as Final Data to Host (FDH) strategy.

In the second strategy depicted in Figure 5(c), instead of reconfiguring to the next temporal partition, on finishing computation for k inputs, we continue the computation on all inputs in each temporal partition. This implies that after the first k computations, we need to store all the intermediate output in the host and then reload the next set of k computations. Once each temporal partition finishes execution on all the input we can proceed to the next temporal partition. This strategy will be beneficial over the FDH method, if the overhead to save and restore the intermediate data is less than the reconfiguration overhead. The reconfiguration overhead in using this strategy is given by $N * C_T + 2 * k * I_{sw} * D_{mem} * \sum_{i=1}^{N} m^i_{temp}$. This technique is referred to as Intermediate Data to Host (IDH) strategy.

The outer loop, I_{sw} will be executed in software. The code for this loop will be generated at the end of the loop fission step. But since the actual value of I will be know only at run time when the input to the problem is known, the actual value of the loop bounds will be filled in at run time. Of course, some care has to be taken here. If I, the total number of inputs to be computed is less than k, the computations done in one run of the RTR system, then only the first k computations from the output will have to be picked up. The software reconfiguration code executing on the host for the FDH strategy would look similar to -

```
for (j=0; j <= I_sw-1; j++) {
 Load  block j  of  input data
 for Configuration 1 into memory.
 for (i=0; i <= N-1; i++) {
   Load Configuration i onto FPGA.
```

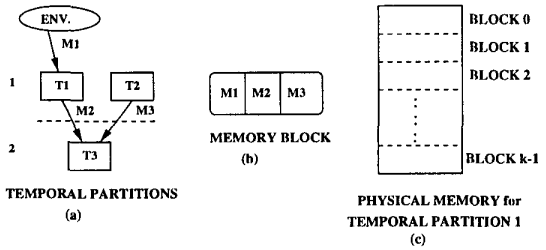

Figure 6: **Placement of memory for different iterations**

```
   Send Start Signal to FPGA.
   Wait for Finish Signal from FPGA.
 }
 Read  block j  of output data
 from memory of Configuration N.
}
```

The software reconfiguration code for the IDH strategy would look similar to -

```
for (i=0; i <= N-1; i++) {
   Load Configuration i onto FPGA.
   for (j=0; j <= I_sw-1; j++) {
      Load  block j of intermediate input data
      for Configuration i into memory.
      Send Start Signal to FPGA.
      Wait for Finish Signal from FPGA.
      Read  block j of intermediate output data
      from memory.
   }
}
```

3 Extensions to HLS for supporting Loop Fission

Once temporal partitioning and loop fission has been performed, the task graph for each temporal partition is to be synthesized by the high level synthesis process. In this section, we discuss some modifications to the traditional synthesis process to adapt it for temporally partitioned designs.

Memory Access Synthesis: In Figure 6(a), we show a task graph mapped to two temporal partitions. The data flowing across temporal partitions needs to be stored in the memory of the reconfigurable processor. All memory segments that are placed in one temporal partition by the temporal partitioning tool for the task graph (without considering the implicit loop) are grouped in one *Memory Block*. There will be k such memory blocks mapped to the physical memory to support the k iterations of the loop. The total amount of intermediate memory for temporal partition 1, m^1_{temp}, shown in Figure 6(a) is equal to M1+M2+M3. The behavioral description of memory access in task T1 is given by -

```
Task 1:
    .
    n = Read(M1[a]) // Read location a in memory M1
    .
    .
    Write(M2[b], z) // Write to location b in
                    // memory  M2
```

All three data flows M1, M2, M3 are placed in the same memory block as shown in Figure 6(b). The actual physical memory for k such blocks corresponding to the memory required to perform all k computations in one temporal partition are shown in Figure 6(c). To access the correct memory block in each iteration, we need to update the code fragment to the following -

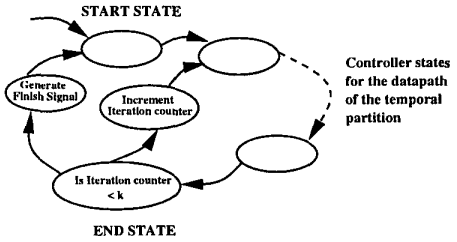

Figure 7: **Augmented Controller for a Temporal Partition**

```
Task 1:
    .
    n = Read(Block[i][offset of M1 in Block + a])
    // Read location a in memory M1
    .
    .
    Write(Block[i][offset of M2 in Block + b], z)
    // Write to location b in memory M2
```

To access memory element at location M1[a] in the i-th iteration of the loop, we need to get to the start of the i-th Memory Block. The offset of memory segment M1 in the memory block will give the starting location of memory segment M1 within the i-th Memory Block. This value added to a will give us the correct memory reference. A memory access can then be synthesized by having an address generation mechanism in hardware which would load the correct address of the memory location to be accessed in a particular iteration. The address generation would generate addresses by the following equation -

*Address = Iteration Index * Size of Block + Offset of Memory M_i within Block + Location within memory M_i*

But since a multiplication operation is expensive, and will increase the area and delay of the synthesized circuit, we round off the memory block in each temporal partition to the nearest power of 2 and perform address generation by a simple concatenation/appending of data values in registers.

Address = Iteration Index.Size of Block + Offset of Memory M_i within Block + Location within memory M_i

The address generation mechanism is simplified, but some memory wastage will occur. This tradeoff between simplifying address generation versus memory wastage has to be made for each RTR architecture. The computation of k, given in Equation 9 has to be changed accordingly.

Controller Synthesis: The software code which has to load the new configuration/next set of memory blocks needs to get a signal from the board upon completion of computation on a temporal partition. This generation of the 'finish' signal needs to be done by the synthesized controller. An iteration counter and a register holding the total iteration value k is required. At the end of a single run of the data path in a temporal partition, the controller would check if the current iteration index of the counter is less than k. If it is, then it increments the counter and goes back to the beginning of the controller states. If it is not, then it generates a 'finish' signal and goes to a start state to wait for a signal from the software to begin execution again. This

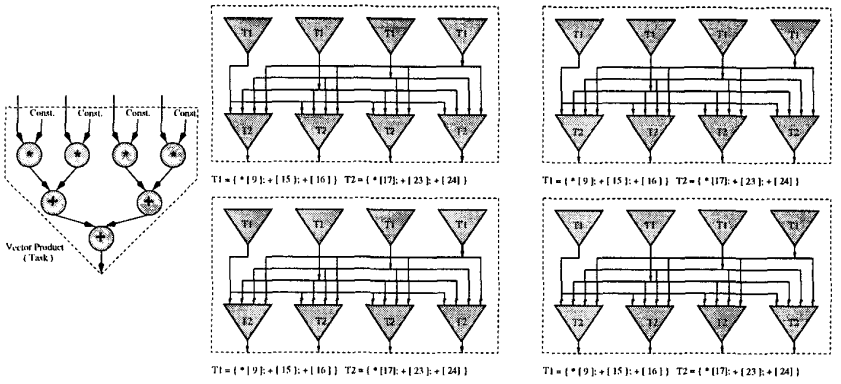

Figure 8: **Task graph for DCT**

augmented finite state machine of the controller for an RTR design is shown in Figure 7.

4 Case Study

We modeled the JPEG image compression algorithm [17], as a Hardware software co-design. The Discrete Cosine Transform (DCT) is the most computationally intensive part of the JPEG algorithm. Therefore DCT was chosen to be implemented in hardware, and the rest of the JPEG subtasks(Quantization, Zig-Zag, and Huffman encoding) were chosen for software implementation. The reconfigurable board consists of a single Xilinx XC4044 FPGA with 1600 CLBs, with a single 64K memory bank with a 32 bit word. The host computer was a Pentium PC with a 200 MHz processor. The host communicates to the reconfigurable board by reading/writing data on the board memory, using a simple handshaking protocol through the PCI bus running at 33MHz. Each reconfiguration of the FPGA takes 100ms. The DCT can be viewed as two consecutive 4x4 matrix multiplications. 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. A vector product is shown in the Figure 8. 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 8 tasks, forms a row of the 4x4 output matrix, as shown in the figure. The entire task graph consists of 4 such collections of tasks.

Two co-design versions of the JPEG compression algorithm were developed. In both cases, the DCT was implemented on the reconfigurable device, while the rest of the JPEG tasks executed in software. In the rest of the discussion, we concentrate on the performance of DCT as a static design and as a dynamic design while ignoring the rest of the tasks, since they have exactly similar execution pattern in both the experiments.

Static Co-design Experiment: For the static design experiment, the board was configured only once at the start of the experimentation with all the computation corresponding to the 4x4 DCT synthesized for the same FPGA. The FPGA could fit two 9 bit multipliers, two 17 bit multipliers, two 16 bit adders and two 24 bit adders. This design used 160 clock cycles with a clock width of 100ns.

RTR Co-design Experiment: The HLS estimator, estimated for tasks of type T1 the FPGA resources to be 70 CLBs. For tasks of type T2, FPGA resources needed are 180

Image	# of 4x4 blocks	I_{sw}	*time in sec* for all blocks in image	
			Static	Dynamic
Parrots	245,760	120	4.13	50.3
Rabbit	194,400	95	3.21	39.9
Grapes	172,800	85	2.88	35.7
Scenery	148,200	73	2.49	30.7
Group	120,000	59	2.03	24.9
XV	90,300	45	1.57	19.0
Small	52,000	20	.95	8.7

Table 1: **DCT execution time for FDH strategy**

CLBs. The ILP model is solved by *CPLEX* software. The result of the model is produced in 3.5 seconds. The temporal partitioning tool divided the task graph into 3 temporal partitions, with all 16 tasks of type T1 on temporal partition 1, 8 tasks of type T2 executed on temporal partition 2 and 8 on partition 3. This is a minimum latency design produced. (A list based temporal partitioner would have placed some tasks of type T2 into temporal partition 1 because it has unused CLBs (1600-60*16 = 640). However doing this would have increased the delay of temporal partition 1, thus increasing the latency of the whole design.)

For throughput maximization, the analyzer tool then looked at all the intermediate data being stored. It calculated that the maximum memory being stored is at the end of temporal partition 1. In partition 1, for a single 4x4 block computation 16 input words are stored and 16 output words are stored as a result of the computation of the 16 tasks placed in partition 1. Therefore for a single computation run, 32 words are stored in partition 1. In partition 2 and 3, 8 words of input and 8 words of output are stored. Therefore we can compute 64k/max(32,16,16) = 2048 blocks of the 4x4 DCT in one run due to the memory limitation of 64k words. The design was then transformed so that it computes 2048 4x4 DCT blocks in one temporal partition. Two different designs were generated, corresponding to the two different RTR sequencing strategies explained earlier in Section 2.2.

To perform a single run of the DCT on a 4x4 block of the input image, after synthesis, the temporal partition 1 needed 68 clock cycles at 50ns, and temporal partitions 2 and 3 needed 36 clock cycles at 70ns. If we ignore the reconfiguration overhead this is a RTR design takes 7560ns less than the static design on a single 4x4 DCT computation. But of course, the reconfiguration overhead is pretty large, i.e., 100ms and we will not see any performance ad-

Image	# of 4x4 blocks	I_{sw}	time in sec for all blocks in image		time in μsec per 4x4 block of DCT		% Improv
			Static	Dynamic	Static	Dynamic	
Parrots	245,760	120	4.13	2.37	16.8	9.6	42
Rabbit	194,400	95	3.21	1.94	16.5	9.9	40
Grapes	172,800	85	2.88	1.76	16.6	10.1	39
Scenery	148,200	73	2.49	1.56	16.8	10.5	37
Group	120,000	59	2.03	1.31	16.9	10.9	35
XV	90,300	45	1.57	1.07	17.3	11.8	31
Small	52,000	20	.95	.74	18.2	14.2	21

Table 2: **DCT execution time for IDH strategy**

vantage unless a large number of computations are done on each temporal partition.

In the strategy where execution of all temporal partitions is performed on the k computations, FDH, we found that even for files of upto 245,760 blocks of DCT computation as shown in Table 1, we did not see any improvement at all. Table 1 shows the image files listed in the decreasing order of their sizes and the total time spent by the RTR and static designs on the images. I_{sw}, is the number of times the RTR sequencer in software has to execute the loop to process all input data on a temporal partition. This was because with a 64k memory bank we can store 2048 blocks of 4x4 DCT, such that k=2048. When we calculated the break even point where the reconfiguration overhead will become smaller than the execution time of the DCT, we will require roughly 42,553 blocks of DCT to be computed in each temporal partition. However since the memory is limited, the dynamic RTR design in this case performs poorly as compared to the static RTR design.

Then we performed computation according to the IDH strategy. The execution time for the DCT, for this experiment is shown in Table 2. The table shows the total time spent by the RTR and static designs on the images. We measured the execution times by inserting probes in the software code at points where the reconfigurable board was invoked to execute the DCT subtask of JPEG. For images requiring 245,760 computations of 4x4 DCT, we show an improvement of 42%. As the size of the image will increase further, more improvement will occur as the reconfiguration overhead will get absorbed.

We conjecture, that if the same experiments are run on reconfigurable devices with less reconfiguration overhead we will see many fold improvement of the RTR design over the static design even for smaller image sizes. For a XC6000 series FPGA, with a reconfiguration overhead of for eg., 500 μ seconds, the improvement we will achieve for the XV file shown in Table 2, is calculated to be 47%.

5 Conclusion

The algorithms presented in this paper are integrated in the SPARCS (Synthesis and Partitioning for Adaptive Reconfigurable Computing Systems) [11, 12] 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] M. J. Wirthlin and B. L. Hutchings, "Sequencing Run-Time Reconfigured Hardware with Software", *ACM/SIGDA International Symposium on Field Programmable Gate Arrays, FPGA 1996, pp. 122-128.*

[2] 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.*

[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] 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.*

[5] K. M. GajjalaPurna and D. Bhatia, "Temporal Partitioning and Scheduling for Reconfigurable Computing", *IEEE Symposium on FPGAs for Custom Computing Machines, FCCM 1998, pp. 329-330.*

[6] J. Spillane and H. Owen, "Temporal Partitioning for Partially-Reconfigurable-Field-Programmable Gate", *Reconfigurable Architectures Workshop in 12th International Parallel Processing Symposium and 9th Symposium on Parallel and Distributed Processing, IPPS/SPDP 1998, pp. 37-42.*

[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] S. Trimberger, "Scheduling designs into a Time-Multiplexed FPGA", *ACM/SIGDA International Symposium on Field Programmable Gate Arrays, FPGA 1998, pp. 153-160.*

[9] S. Trimberger, "A Time-Multiplexed FPGA", *IEEE Symposium on FPGAs for Custom Computing Machines, FCCM 1997, pp. 22-28.*

[10] M. Xu, F. Kurdahi, "Layout Driven High Level Synthesis for FPGA Based Architectures", *Design and Test in Europe '98.*

[11] 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. 31-36.*

[12] 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.*

[13] 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.*

[14] M. Wolf, *High Performance Compilers for Parallel Computing*, Addison-Wesley Publishers, 1996.

[15] C. H. Gebotys, "An Optimal methodology of Synthesis of DSP Multichip Architectures", *Journal of VLSI Signal Processing, v11, p9-19 1995.*

[16] R. Niemann and P. Marwedel, "An Algorithm for Hardware/Software Partitioning Using Mixed Integer Linear Programming", *Proceedings of the ED&TC, 1996.*

[17] G.K. Wallace, "The JPEG Still Picture Compression Standard", *ACM Communications, 1991.*

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