An Integrated Temporal Partitioning and Partial Reconfiguration Technique for Design Latency Improvement *

Satish Ganesan and Ranga Vemuri
Department of ECECS, ML 0030
University of Cincinnati,Cincinnati, OH 45221.
{*satish,ranga*}*@ececs.uc.edu*

Abstract

Partially reconfigurable processors provide the unique ability by which a part of the device can be reconfigured, while the remaining part is still operational. In this paper, we present a novel partitioning methodology that temporally partitions a design for such a partially reconfigurable processor and improves design latency by minimizing reconfiguration overhead. This is achieved by overlapping execution of one temporal partition with the reconfiguration of another, using the processors partial reconfiguration capability. We have incorporated block-processing in the partitioning framework for reducing reconfiguration overhead of partitioned designs. A highlight of our partitioner is it's ability to handle loops and conditional constructs in the input specification. The proposed methodology was tested on several examples on the Xilinx 6200 FPGA. The results show significant reduction in the design latency, leading to a considerable speed-up due to partial reconfiguration.

1 Introduction

Dynamically reconfigurable processors have the potential for achieving high performance at a relatively low cost for a wide range of applications. Reconfigurable devices, such as Field Programmable Gate Arrays (FPGA), can also implement large designs by the virtue of partitioning the design in time [1, 2] leading to run-time reconfigurable implementations of the design. However, the reconfigurable processors typically have a high reconfiguration overhead, which degrades the performance of the design.

Certain partially reconfigurable processors [3, 4] possess the unique capability by which a part of the device can be operational while the remaining part is being reconfigured. This feature can be used to overlap execution and reconfiguration of different portions of the design leading to partial, if not complete, amortization of the reconfiguration overhead and significant improvement in the design latency. This advantageous feature of such partially reconfigurable computing (PRC) systems motivates the work in this paper. We propose a novel technique to generate RTR designs for a PRC device, that improves design latency by reduction of the reconfiguration overhead posed by the device. A highlight of our partitioner is the capability to handle control constructs in the input specification.

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

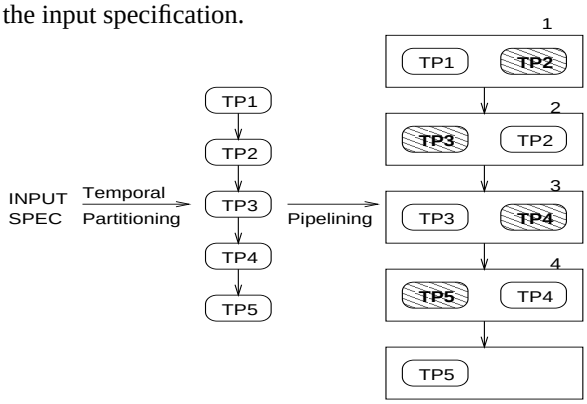

Figure 1. Partitioning/Pipelining Methodology

Figure 1 depicts the overview of our approach. The first step is to *partition* the design into a sequence of temporal segments. This is followed by a *pipelining* phase, where the execution of each temporal partition is pipelined with the reconfiguration of the following partition. Referring to Figure 1, at the i^{th} instant, TP_i executes and TP_{i+1} reconfigures on the PRC device. Reconfiguration time of segment TP_{i+1} is reduced due to overlap with execution of TP_i. Similarly, the $(i+1)^{th}$ instant involves overlap of execution of TP_{i+1} and reconfiguration of TP_{i+2}. This process is continued until all the temporal segments have been loaded and executed.

Let R_i and E_i be the reconfiguration and execution time respectively, of the i^{th} TP segment on the target architecture. The total latency of the design using our proposed technique is:

$$Lat_n = R_1 + \sum_{i=1}^{n-1} max(R_{i+1}, E_i) + E_n \quad (1)$$

Hence when

$$R_{i+1} - E_i \leq 0 \;\; \forall i \;\; 1 \leq i \leq n-1 \quad (2)$$

there is complete amortization of the reconfiguration overhead using partial reconfiguration. Hence, it is clear that in order to obtain significant improvement in design performance, the reconfiguration time of TP_{i+1} should be comparable to the execution time of TP_i. This allows maximal *overlap* between execution and reconfiguration and results in considerable reduction in reconfiguration overhead. When device reconfiguration times are much higher than design execution times, it becomes essential to group computationally intensive structures, e.g. loops, in a single temporal segment to increase E_i and thereby minimize $R_{i+1} - E_i$.

The remainder of this paper is organized as follows. In Section 2, we discuss related work, and in Section 3 we detail our PRC synthesis framework. In Section 4, we describe our partitioning algorithm and in Section 5, the reconfiguration-execution pipelining strategy. In Section 6, we discuss block processing in conjunction with our approach. Section 7 presents results and in Section 8, we provide our concluding remarks.

2 Related Work

Luk et al. [5] have proposed a strategy to combine two or more designs into one reconfigurable design, based mainly on the identification of components common to these designs. They perform partial reconfiguration to reduce the reconfiguration overhead. Turner and Woods [6] have suggested a method to form reconfiguration sets by maximizing the static hardware between reconfigurations and achieve speed-up using partial reconfiguration. Schwabe et al. [7] take advantage of some of the features of the Xilinx XC6200 family of FPGAs to reduce the reconfiguration time overhead by compression of the configuration bit streams.

Though the above methods proposed to reduce reconfiguration overhead are sound, the authors assume that designs are small and would fit on a single processor. However, many of the designs are too large to be placed in a single device. The approach we propose takes a single design and improves its overall latency by coupling it with reduction in the processor's reconfiguration overhead. We address the problem as a structure-oriented partitioning problem that reduces reconfiguration overhead between successive partitions.

3 PRC Synthesis Framework

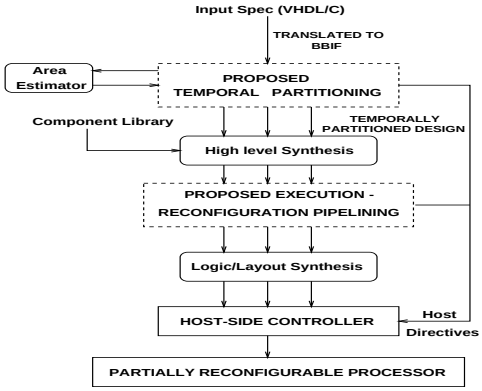

Figure 2. PRC Synthesis Framework

Figure 2 depicts our synthesis framework. The input to the framework is a behavioral specification in VHDL or C. The specification is translated into an intermediate Control Data Flow Graph (CDFG) representation and fed to the temporal partitioner. The partitioner, with the help of a behavioral area estimator, produces a sequence of temporal segments that implements the design. High-level synthesis [8] is performed on each temporal partition to produce a register transfer level (RTL) implementation of the design. Logic and layout synthesis generates bitmap files of the RTL temporal partitions. A host-side controller loads and executes the synthesized partitions on the PRC device. In the following subsections, we detail our input model, the target architecture model and the host-PRC interaction model.

3.1 Input Specification

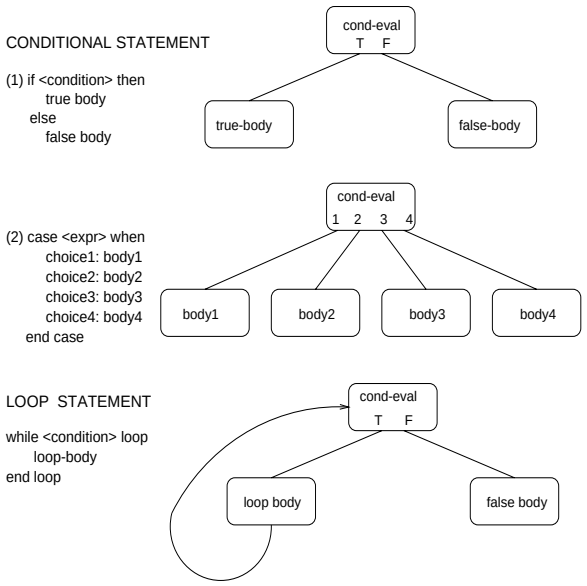

Figure 3. Control Constructs in BBIF

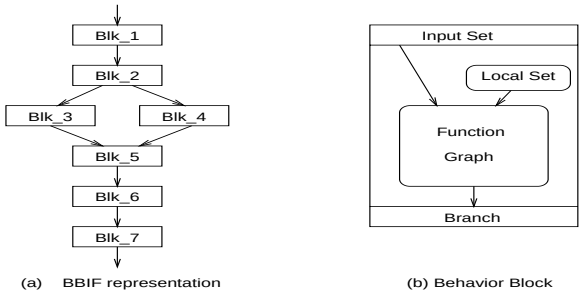

Figure 4. Behavior Block in BBIF

Our input model, called the Behavior Blocks Intermediate Format (BBIF) [9], is a Control Data Flow Graph (CDFG) extracted from the behavior specification. A conditional statement translates into one block (that evaluates the conditional predicate) and a collection of block branches, one for each branch body. Similarly, a loop statement is realized by a behavior block that implements the loop predicate evaluation followed by a conditional branch either to the loop body or outside the loop.

Figure 4 illustrates a typical behavior block. A block consists of an set of input carriers, a set of local carriers, a set of functions (operations) and a set of output carriers. The function graph is a directed acyclic graph that captures the data flow. Data flow between blocks happens strictly through branch interface.

Note that the BBIF model denotes a *single thread of control*. In other words, at any time only one of the blocks will be exe-

cuting. We plan to extend our methodology to multi-threaded specifications, where more than one block can execute at the same time.

3.2 Target Architecture Model

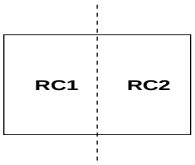

Figure 5. Target Architecture

The target architecture model is illustrated in Figure 5. The PRC device is split into two parts, RC_1 and RC_2. The design is implemented on the RC such that, when TP_i executes on RC_1, TP_{i+1} reconfigures on RC_2. Similarly, when TP_{i+1} executes on RC_2, TP_{i+2} reconfigures on RC_1. The PRC device is divided into two parts as at any time there exists only two active events on the device at a given time: execution and reconfiguration. RC_1 and RC_2 have a fixed area and position on the PRC device. This ensures that a TP on either of these parts remains undisturbed between the time it reconfigures and executes, thereby aiding partial reconfiguration.

3.3 Host-PRC Interaction Model

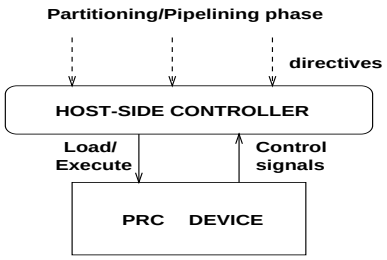

Figure 6. Host-PRC Interaction Model

A *host-side controller* handles the interaction between the PRC device and the host processor. The host processor is responsible for loading and executing the partitioned modules on the PRC device. The host-side controller provides a handshaking protocol between the host and the device. The controller is a finite state machine (FSM), where in every state the partitioned modules are either loaded, executed or no operation is performed on them. State transitions are based on control signals obtained from the design executing or reconfiguring on the PRC device. The FSM is derived from a set of *host-controller semantics* that are generated by the partitioning and pipelining phases. These directives are explained in more detail along with the algorithm in the following section.

4 Temporal Partitioning

Given the block graph specification, the partitioner has to temporally partition the graph into k segments such that:

(1) area(TP_i) $\leq$ area(RC_1) $\forall$ odd i $1 \leq i \leq k$
(2) area(TP_i) $\leq$ area(RC_2) $\forall$ even i $1 \leq i \leq k$
(3) $\exists$ no loops across TPs

When a design is too large to fit on the PRC device, the procedure *Temporal_Partitioning(BBIF, blk_area, prc_area)* traverses the block graph and performs appropriate actions based on the area and the block type of individual blocks, which can be either a loop block (Lblk), conditional block (Cblk), or a non-control construct block.

The procedure *Partition_Block(blk)* is invoked to partition the operation graph of a block when the estimated area of that block violates the area constraint imposed on the partitioner. Any operation graph partitioning algorithm can be used for this purpose. The conditions imposed on such a partitioner is as follows: (1) The partitioner should not introduce cycles in the block graph (2) The partitioner should minimize the average number of data transfers between partitions.

The block–partitioner generates a sequence of acyclic partitioned segments. The next block that is traversed by the procedure *Temporal_Partitioning(BBIF, blk_area, prc_area* is the last segment in the sequence of the partitioned segments. The last segment in the sequence is assigned the same type as the original block. This will ensure that if the type of the original block is *Lblk* or *Cblk*, the corresponding procedures are invoked based on the type. The block–partitioning scenario is illustrated in Figure 8.

The procedure *Handle_Loop* is invoked when a block of type *Lblk* is encountered in the BBIF block graph. The procedure obtains the cumulative area of all the blocks in the entire loop structure using an area estimator. If the estimated area meets the area constraint, all the blocks in the loop are merged into a single partition. If the area constraint cannot be met, the exception is handled by grouping all the blocks in the loop structure so that the loop fits on the entire PRC device. *Exception handling* is done to accommodate large loop bodies in the input specification. If the loop does not fit on the entire device either, the partitioner reports a failure as otherwise the loop has to be partitioned across temporal segments.

If the block type encountered is a *Cblk* block, the procedure *Handle_Conditional* obtains the area of all the branches of the conditional evaluating block. If the estimated area meets the partitioner's area constraints, these blocks are grouped into a single partition. If the area constraint is violated, a *host polling* strategy is adopted. Performing *Partition_Block* before handling conditionals ensures that if a conditional block is too large to fit on a device partition, it is partitioned into smaller blocks before *Handle_Conditional* is invoked on that block.

The effect of the partition methodology described above on the associated execution model is detailed in the following sub–sections.

4.1 Execution Model for Loop Handling

When a *Lblk* structure is encountered in BBIF, the entire loop is grouped into a single temporal partition if the area constraint is not violated. This ensures that the corresponding temporal partition spends a significant amount of time in execution. The execution time can be maximally over-

```
Algorithm: Temporal_Partitioning(BBIF, blk_area, prc_area)
  Input: BBIF: the input block graph,
    blk_area: area constraint on TP_i,
    prc_area : area of the PRC device
  Output: Partitioned block graph
begin
    area ← {estimated area of block graph}
    if (area > prc_area) then
        ▷ for loop traverses block graph
        for each blk in block_graph
            block_type ← {type of the current block}
            area ← {estimated area of current block}
            if (area > blk_area) then
                Partition_Block(blk)
            end if
            if (block_type is Lblk) then
                Handle_Loop(blk, blk_area, prc_area)
            else if (block_type is Cblk) then
                Handle_Conditional(blk, blk_area)
            end if
        end for
    end if
end
```

```
Algorithm: Handle_Loop(blk, blk_area, prc_area)
  Input:
    blk: a loop block in the BBIF graph,
    blk_area: area constraint on TP_i,
    prc_area: area of PRC device
  Output: Merged Loop Structure
begin
    Lset ← {set of all blocks in loop}
    area ← {estimated area of Lset}
    if (area < blk_area) then
        ▷ here, group loop in a single partition
        merge_all_blocks(Lset)
    ▷ here, perform exception handling for loops
    else if (area < rc_area) then
        ▷ here, group loop on the entire device
        merge_all_blocks(Lset)
    else
        report_failure
    end if
end
```

```
Algorithm: Handle_Conditional(blk, blk_area)
  Input:
    blk: a conditional block in the BBIF graph,
    blk_area: area constraint on TP_i
  Output: Merged Conditional Branches
begin
    Cset ← {set of all branches of the conditional block}
    area ← {estimated area of Cset}
    if (area < blk_area) then
        ▷ here, group branches in a single partition
        merge_all_blocks(Cset)
    else
        ▷ here, host waits for conditional predicate evaluation
        adopt_host_polling_strategy
    end if
```

Figure 7. Partitioning Algorithm

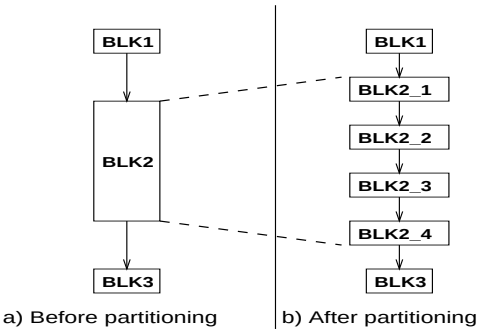

Figure 8. Partitioning Large Blocks

```
        ...
partition TP4
    area occupied (part —— device) : part
    host poll on ( signal —— null) : flag
end TP4
partition TP5
    area occupied (part —— device) : device
    host poll on ( signal —— null) : null
end TP5
partition TP6
        ...
```

Figure 9. Host Controller Semantics: Partitioning

lapped with the reconfiguration of the following temporal partition to reduce it's reconfiguration overhead. When the loop executes, the next partition is reconfigured on the PRC device. Execution of the loop partition completes when the loop predicate evaluates to false. The corresponding *host-side controller semantics* generated as by–products of partitioning is shown in Figure 9.

When the entire loop cannot be grouped in a single partition, it is possible that the loop fits on the entire PRC device. Two strategies can be adopted in this case. The first one is to report a failure as the loop violates area constraints . A second strategy would be to handle the exception by loading the loop structure on the entire PRC device rather than reporting a failure. The next temporal segment in this case is loaded only when the loop has completed execution. No overlap of execution and reconfiguration is possible ($R_{i+1} - E_i = R_{i+1}$, where R_i and E_i are the reconfiguration and execution times of TP_i) in this case. In our framework, we follow the second strategy to accommodate large loop bodies in the input specification.

4.2 Execution Model for Conditional Handling

A *Cblk* block in BBIF branches to two or more blocks in the block graph. However, due to the single thread of control, only one of these branches executes on the PRC. The host should therefore load the appropriate branch depending on the outcome of the conditional predicate evaluation. We call this strategy the *host polling* strategy. In this case, the reconfiguration of the branch cannot be overlapped with the execution of the condition evaluating block and this results in a reconfiguration overhead for the branch block ($R_{i+1} - E_i = R_{i+1}$, where R_i and E_i are the reconfiguration and execution times of TP_i). In order to achieve some amount of overlap, we can group all the conditional branches into a single temporal partition, provided the area constraint is met. Since all

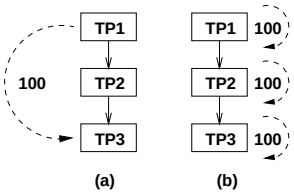

Figure 10. Block-Processing

the branches are reconfigured, the host need not wait on the condition to evaluate. This ensures that the reconfiguration of the branch overlaps with some amount of execution of the condition evaluating block ($R_{i+1} - E_i$ is minimized). In our framework, the partitioner attempts to group the branches into a single partition to reduce reconfiguration overhead. However, if the grouping fails, the *host polling* strategy is adopted.

5 Partitioning and Block-Processing

As we noticed before, reconfiguration overhead becomes insignificant if execution time of a partition is comparable to the reconfiguration time of the following partition. High execution times are possible, if a partition can execute a loop. In many application domains like Digital Signal Processing, computations can be repeatedly performed on long streams of input data. For example, a 4x4 FFT is usually performed on hundreds of 4x4 input matrices and not on just one input matrix. This approach known as *block-processing* has been used to increase system throughput in the area of parallel compilers [10], VLSI processors [11] and temporal partitioning [12].

Figure 10 illustrates the use of block-processing in conjunction with our partitioning methodology. The partitioner produces 3 temporal partitions. The design has to be executed for 100 input data streams which implies that the design has to be executed 100 times. Let us assume that the execution time for each TP is 5 μs and the reconfiguration time is 500 μs. Therefore, the reconfiguration overhead for TP_2 will be 455 μs (5 μs is saved by overlap with the execution of TP_1). However, if we perform block-processing by sequencing all 100 computations on each temporal partition, the total execution time of each TP is now 500 μs. Reconfiguration overhead of TP_2 is now reduced to 0. Thus block-processing amortizes the reconfiguration overhead over 100 inputs. Block-processing is possible only for applications that process large streams of input data. We represent such applications by a graph having an implicit outer loop as shown in Figure 10a. Note that block-processing is possible if there are no dependencies among computations for different inputs. This means there should be no loop-carry dependencies due to the implicit outer loop among different iterations of the loop. Most of the DSP applications fall in this category.

For this approach, Equation 3 can now be rewritten as: When

$$R_{i+1} - m * E_i \leq 0 \quad \forall i \; 1 \leq i \leq n-1 \tag{3}$$

there is complete amortization of the reconfiguration overhead; where m is the number of input data streams for which computation is performed.

6 Results

The proposed algorithms were integrated in the SPARCS environment (Synthesis and Partitioning for Adaptive Reconfigurable Computing Systems) [13]. For obtaining results, we used the following benchmarks: 4x4 Discrete Cosine Transform (DCT), 2 dimension 4x4 Fast Fourier Transform (FFT), 16-tap Finite Impulse Response Filter (FIR), a Synthetic Example (SEG), 1 dimension 4x4 FFT and the Traffic Light Controller (TLC).

High-level synthesis was performed using ASSERTA [9]. VELAB [14] was used to perform logic elaboration at RT level and XACT6000 tools were used for place and route. Due to the poor quality of the available tools, only a small portion of the total area could be used for logic to enable successful routing. The area constraint given to the partitioner took the routing overhead into account.

Table 1 provides the comparative results obtained by using the partial reconfiguration (PR) technique proposed in this paper (refer Figure 7) versus the full reconfiguration (FR) technique. For the FR technique, the only step performed is temporal partitioning with the area constraint being the area of the entire device. The table lists the following data: #TP: number of temporal partitions, #Inp blocks: number of input streams of data (this will be 1 if block processing is not used), Rec. time: total time spent only in configuration/reconfiguration in μsecs, Exec. time: total time spent in execution in μsecs, Throughput: total time taken to process all data in μsecs ($rec.time + exec.time$), %rec - percentage of the total time spent in reconfiguration ($rec.time/throughput * 100$), Speed-up: factor by which latency improves ($throughput(FR)/throughput(PR)$). Reconfiguration for the PR approach was performed using a 33 MHz PCI bus and the design clock frequency was 16 MHz for all these benchmarks. In the case of the FR approach, the results were obtained using a 10 MHz clock. The reduced clock speed for the FR approach is due to the fact that the area of the partitions generated are larger, which impacts placement and routing, which in turn determines the clock speed.

We observe from Table 1 that following our methodology results in good improvement in the design throughput as compared to the FR approach. Block-processing was followed for the DCT, 1d-FFT, 2d-FFT and FIR examples for both the approaches. For DCT, total reconfiguration time was reduced to 2.52% of the total latency using the our approach and resulted in a speed up factor of 2.0. The 2d-FFT, FIR and 1d-FFT benchmarks also showed significant improvements in design throughput and considerable reduction in the reconfiguration overhead. The SEG consisted of a sequence of cascaded loops. The partitioner grouped each of these loops into separate partitions. The total reconfiguration time for the SEG was reduced to 30% using our approach as compared to the 61.3% using FR. Also, a speed up of 1.8 over

Design	Method	# TP	# Inp Blocks	Rec. time (μs)	Exec. time (μs)	Throughput (ms)	% rec.	Speed-up vs full
DCT	PR	48	180	51.47	1991	2.04	2.52	2.0 x
	FR	22	180	1995.7	2088	4.08	51.2	
SEG	PR	3	1	165	385	550	30	1.8x
	FR	2	1	610	385	995	61.3	
2d FFT	PR	9	140	56.8	446.25	0.5	11.4	1.7x
	FR	4	140	456.6	245	0.7	65.2	
FIR	PR	6	150	56.6	273.7	0.33	17.2	1.71x
	FR	3	150	274.9	198.2	0.47	58.04	
1d FFT	PR	3	165	54.03	154.7	0.21	25.7	1.52x
	FR	2	165	154.8	103.1	0.26	59.5	
TLC	PR	1	1	86	0.9	86.9	98.9	1x
	FR	1	1	86	0.9	86.9	98.9	

Table 1. Latency improvement and Reconfiguration Time Reduction using PR approach

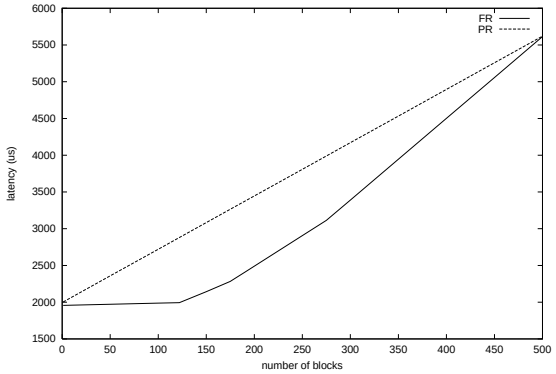

Figure 11. 1d DCT: Throughput vs #i/p blocks for FR and PR

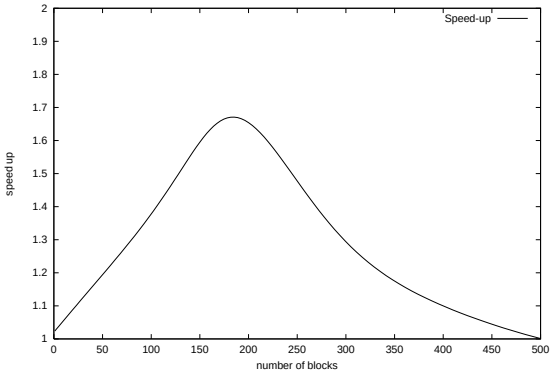

Figure 12. 1d DCT: Speed-up vs #i/p blocks

the FR approach was achieved. The final example, TLC, fit in a single device. Hence no partitioning was required and therefore results for both approaches remain the same.

Figure 11 presents the variation of the latency for different values of the number of input blocks for both the PR and FR approaches. Figure 12 shows the corresponding speed-up of PR over FR. We observe that when the number of blocks processed is low, the speed-up achieved is close to 1. This is because, a major portion of the time is spent in reconfiguration for both the approaches. The amount of overlap achieved using the PR approach is negligible. When a very large number of input blocks are processed, reconfiguration time is very less when compared to the total execution time for both the approaches. The maximum speed-up was achieved when the number of input blocks was around 180.

7 Concluding Remarks

In this paper, we have presented a novel temporal partitioning, and execution and reconfiguration pipelining technique to partition designs for partial RC systems. The goal was to partition such that reconfiguration overhead posed by the processor is minimized and design latency is improved. The partitioning was performed with the knowledge of the structure of the input specification. We also proposed performing block-processing along with partitioning to improve design performance. The experimental results obtained demonstrate the effectiveness of our technique. We observed that capturing loop structures into partitions improved design throughput significantly.

References

[1] M.Kaul and Ranga Vemuri, "Temporal Partitioning combined with Design Space Exploration for Latency Minimization of Run-Time Reconfigured Designs", Design and Test in Europe, DATE 98, IEEE Computer Society, Paris, 1998 pp.389-396.

[2] K. M. GajjalaPurna and D. Bhatia, "Partitioning in Time: A Paradigm for Reconfigurable Computing", ICCD98, IEEE Computer Society, October, 1998, pp. 340-345.

[3] Xilinx Corporation, San Jose, California, XC6200 Datasheet, 1997.

[4] Atmel Corporation, San Jose, California, http://www.atmel.com.

[5] N. Shirazi, W. Luk, "Automating Production of Run-Time Reconfigurable Designs", Field-Programmable Gate Arrays, FPGA 1996, pp.147-156.

[6] J-P Heron, R.F. Woods, "Accelerating run-time reconfiguration on FCCMs", IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM '99, Preliminary Proceedings, Napa, CA, April 21-23, 1999.

[7] S. Hauck, Z. Li, E. J. Schwabe, "Configuration Compression for the Xilinx XC6200 FPGA", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 18, No. 8, August, 1999, pp. 1107-1113.

[8] D. D. Gajski, N. D. Dutt, A. Wu, S. Lin, High-level Synthesis: Introduction to Chip and System Design, Kluwer Academic Publishers, 1992.

[9] N. Narasimhan "Formal Synthesis: Formal Assertions Based Verification in a High-Level Synthesis System", PhD Thesis, University of Cincinnati, 1998.

[10] M. Wolf, High Performance Compilers for Parallel Computing, Addison-Wesley Publications, 1996.

[11] S. Y. Kung, VLSI Array Processors, Prentice Hall, 1988.

[12] M. Kaul, R. Vemuri, "Integrated Block-Processing and Design-Space Exploration in Temporal Partitioning for RTR Architectures", Reconfigurable Architectures Workshop, RAW'99, Springer Publ., pp.606-615.

[13] 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, RAW'98, Springer Publ., pp.31-36.

[14] Xilinx Corporation, San Jose, California, VELAB Reference Manual, 1998.