# Temporal Partitioning and Scheduling Data Flow Graphs for Reconfigurable Computers

Karthikeya M. Gajjala Purna, *Student Member*, *IEEE* and Dinesh Bhatia, *Member*, *IEEE*

**Abstract**—FPGA-based configurable computing machines are evolving rapidly. They offer the ability to deliver very high performance at a fraction of the cost when compared to supercomputers. The first generation of configurable computers (those with multiple FPGAs connected using a specific interconnect) used statically reconfigurable FPGAs. On these configurable computers, computations are performed by partitioning an entire task into spatially interconnected subtasks. Such configurable computers are used in logic emulation systems and for functional verification of hardware. In general, configurable computers provide the ability to reconfigure rapidly to any desired custom form. Hence, the available resources can be reused effectively to cut down the hardware costs and also improve the performance. In this paper, we introduce the concept of temporal partitioning to partition a task into temporally interconnected subtasks. Specifically, we present algorithms for temporal partitioning and scheduling data flow graphs for configurable computers. We are given a configurable computing unit (RPU) with a logic capacity of $S_{RPU}$ and a computational task represented by an *acyclic data flow graph* $G = (V, E)$. Computations with logic area requirements that exceed $S_{RPU}$ cannot be completely mapped on a configurable computer (using traditional spatial mapping techniques). However, a *temporal partitioning* of the data flow graph followed by proper scheduling can facilitate the configurable computer based execution. Temporal partitioning of the data flow graph is a $k$-way partitioning of $G = (V, E)$ such that each partitioned segment will not exceed $S_{RPU}$ in its logic requirement. Scheduling assigns an execution order to the partitioned segments so as to ensure proper execution. Thus, for each segment in $\{s_1, s_2, \cdots, s_k\}$, scheduling assigns a unique ordering $s_i \rightarrow j$, $1 \leq i \leq k$, $1 \leq j \leq k$, such that the computation would execute in proper sequential order as defined by the flow graph $G = (V, E)$.

**Index Terms**— Configurable computing, field programmable gate arrays, spatial partitioning, temporal partitioning, scheduling, data flow graphs, reconfigurable computers, high performance computing.

## 1 Introduction

The configurable computing paradigm [19] is a hybrid of the two traditional computing paradigms: general purpose computing and application specific computing (ASC). General purpose computing [37], [13] is defined around instruction set architecture(microprocessor) machines. This paradigm has long been in existence and is very popular primarily because of its ease of use. General purpose computers are facilitated by software tools like compilers [1] that facilitate easy and effective mapping of applications. Application specific computing (ASC) supports customization of applications in the form of hardware. Due to customization of hardware, this approach offers maximum performance for executing applications. This approach is suitable for acceleration of compute intensive (e.g., signal and image processing) applications that take a long time to execute on general purpose machines. However, due to the need for customization, this approach is not very flexible and has tremendous cost overheads.

Configurable computing has the potential of offering the performance that is comparable to that of custom hardware and a flexibility comparable to that of a general purpose machine. Configurable computing machines are built around programmable hardware, essentially consisting of fine or coarse grain programmable devices called Field Programmable Gate Arrays (FPGAs) [8], [21]. Typically, they consist of a collection of FPGAs interconnected using a fixed or programmable interconnect [9], [20]. The majority of the configurable computing machines [23], [35], [7], [18], [43] operate as coprocessors [2] and have some local memory. Fig. 1 shows a model under which most machines operate. The coprocessors can be, on demand, configured to perform any desired function.

Configurable computing machines have been demonstrated to have a wide variety of applications from logic emulation [4], [44], [40], [11], [39] to algorithm specific hardware execution. Some of the compute intensive tasks that have made effective use of configurable computing machines include DNA sequence matching [22], RSA cryptography [23], text searching [27], fingerprint matching [24], and high-speed image processing [3].

A custom computing machine and its architecture are static in nature, i.e., in no instance can the size of the mapped application exceed the size (or capacity) of the programmable hardware. This capacity/size is usually measured in terms of gate equivalence [48]. Large scale FPGA-based compute engines include logic emulators and hardware accelerators.

FPGA-based logic emulation is important for functional verification of large designs. As the design complexity grows, emulation becomes a key factor in the rapid prototyping of large scale designs. Several academic [4]

• *K.M. Gajjala Purna is with Synopsys Inc., 700 E. Middlefield Rd., Mountain View, CA 94043-4033. E-mail: kgajjala@synopsys.com.*
• *D. Bhatia is with the Design Automation Laboratory, Electrical and Computer Engineering and Computer Science Department, University of Cincinnati, Cincinnati, OH 45521-0030. E-mail: dinesh.bhatia@uc.edu.*

*For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 109710.*

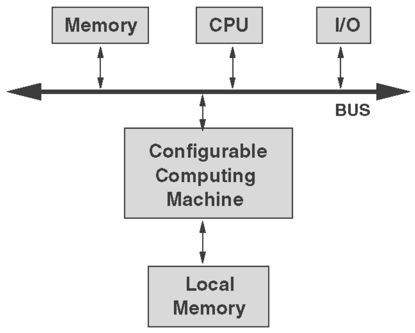


Fig. 1. Typical model for configurable computing machines.

and commercial logic emulators [40], [11], [39] have come into existence. Typically, these systems are built with large scale programmable hardware. As the size of the programmable hardware increases, the cost of the associated system also scales up tremendously. There is a limitation on the size of the design/application that can be mapped onto a configurable machine. In order to effectively use a multidevice reconfigurable architecture, a collection of CAD tools is required. One of the more important CAD tools is the multidevice partitioner. A multidevice partitioner must operate under area (size of each programmable device) and pin (number of programmable I/Os per device) constraints. Generally, the pins are consumed much faster than the logic during multidevice partitioning. Thus, few devices yield small logic density, while a large number of devices result in poor logic utilization (per device) and lower performance. Hence, a suitable size for a configurable computing machine is one that results in best area to performance trade-off. In fact, very large architectures yield poor logic utilization. Thus, the cost of the architecture does not scale linearly with the size of the application. It must factor in poor logic utilization as design sizes increase. Finally, the size of the largest application that can be implemented by programmable hardware is limited by the physical size of the hardware itself.

In this paper, our interest lies in hardware implementation of applications that have logic requirements that greatly exceed the logic capacity of the configurable computer. In order to facilitate the execution of such an application, we have proposed a temporal partitioning based mapping methodology. Temporal partitioning divides the design into mutually exclusive, limited size segments such that the logic requirement for implementing a segment is less than or equal to the logic capacity of the configurable computer. Such temporal segments can be scheduled for execution in proper order to ensure correct overall execution.

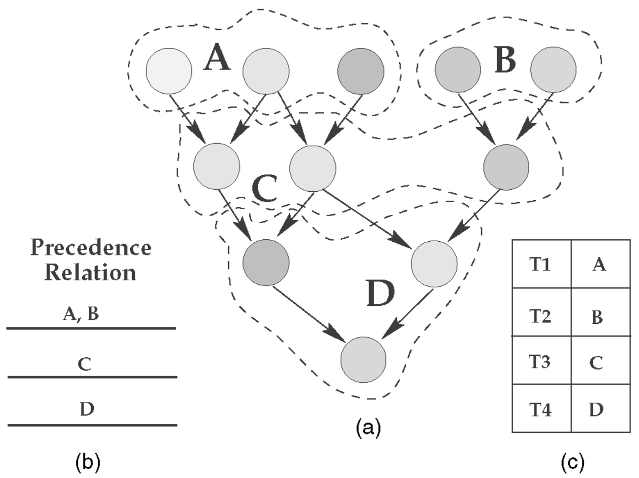


Fig. 2. An example data flow graph (DFG) with its partitioning and scheduling.

## 2 Motivating Example

We illustrate our concept with the help of a motivating example. Fig. 2 illustrates an example where an application expressed in terms of a data flow graph (DFG) is partitioned into four (labeled A, B, C, and D) segments. For such a graph, each node represents an operator or a function. If the directed edges are assumed to represent the data dependency between the nodes of the graph, then it is clear that segments A and B have no interdependency. Segment C depends on the outputs from segments A and B, and segment D depends on the outputs of segment C. Thus, it is possible to execute the entire application on a limited size hardware if it is ensured that segment A will execute prior to segment C, segment B will execute prior to segment C, and segment C will execute prior to segment D. Fig. 2c illustrates one such mapping where segments A, B, C, and D are mapped to time steps $t_1$, $t_2$, $t_3$, and $t_4$, respectively, and $t_i$ precedes $t_j$ if $i < j$.

In order to ensure proper execution of the application represented by the graph in Fig. 2, the output generated by the execution of segment A must be stored in a buffer so that it is available(as an input) when segment C is executed. Similar arguments for data buffering hold for other segments also.

**Definition 1.** *An input value to a segment is called* `stable` *if it is also the value that would result from a correct implementation of the entire application on one large hardware.*

For example, in Fig. 2, the inputs to segment C are `stable` only after segments A and B have finished executing. Prior to execution of segment A, any value at the input of segment C is unstable.

**Definition 2.** *A* `precedence-relation` *defines an ordering of the partitioned segments that would guarantee a* `stable` *input for each partition.*

Fig. 2b illustrates a `precedence-relation` for the segments defined in Fig. 2a. Note that segments A and B are not dependent on each other and thus can be executed in any order with respect to each other.

## 3 Related Work

Partitioning and scheduling of graphs, with minor variations in the objective being pursued, is an active problem in many areas of research. Some of the research areas that address this problem include multiprocessing

[16], [33], high level synthesis [17], [25], and reconfigurable computing.

In multiprocessing, *partitioning* and *scheduling* are necessary for the programs to exploit parallelism. Partitioning ensures that the granularity of the parallel program is coarse enough for the target multiprocessor without losing too much parallelism. Scheduling is necessary to achieve good processor utilization and to optimize interprocessor communication in the target multiprocessor. Thus, in the presence of two or more processors, our example in Fig. 2 would dispatch (for simultaneous exection) segments A and B onto two independent processors.

*Partitioning* and *Scheduling* in high level synthesis is performed under resource constraints [32] and/or timing constraints [26]. The high level synthesis tools perform partitioning and scheduling for a fixed datapath architecture. The design choices made in the datapath have a direct impact on the complexity of the control path to be synthesized. In the case of reconfigurable computers, the architecture can change from one configuration to another. This adaptive nature of reconfigurable computers yields better performance and makes the synthesis task easier.

Several novel reconfigurable architectures have been proposed around the concept of time sharing logic resources. These include Time Multiplexed FPGA [42], Dharma [6], and DPGA [14]. The architectures require specialized CAD tools for physical mapping, as proposed in [41], [5], and [15].

In this research, we have developed methodologies to overcome mapping problems associated with area constrained hardware. Specifically, we have developed algorithms for temporal partitioning and scheduling of large designs/applications on area constrained reconfigurable hardware. The algorithms are very generic in nature and can be easily targeted toward any reconfigurable architecture that resembles the abstract model described in Section 1. The overall mapping process requires *temporal partitioning* followed by proper scheduling of application segments (application segments that result from temporal partitioning). Each schedule executes under a control machine which ensures stable inputs to each temporal segment in the schedule.

## 4 Problem Formulation

A program or an application is represented by a DFG. A DFG is a directed acyclic graph, $G = (V, E, W, D)$, where $V$ is a set of nodes, $|V| = n$, and $E$ is a set of edges. For each node $v_i \in V$, there exists a weight $w_i \in W$, $1 \leq i \leq n$, and delay $d_i \in D$, $1 \leq i \leq n$. Each node $v_i \in V$ represents an implementation of a functional operation and, correspondingly, $w_i$ represents the size of the logic and $d_i$ represents the delay of the function.[1] A directed edge $e_{ij} =< v_i, v_j >$, $e_{ij} \in E$ exists *iff* the function represented by $v_j$ depends on the output of the function represented by $v_i$. A large number of compute intensive signal and image processing applications can easily be incorporated in such a model.

The problem of partitioning can be stated as:

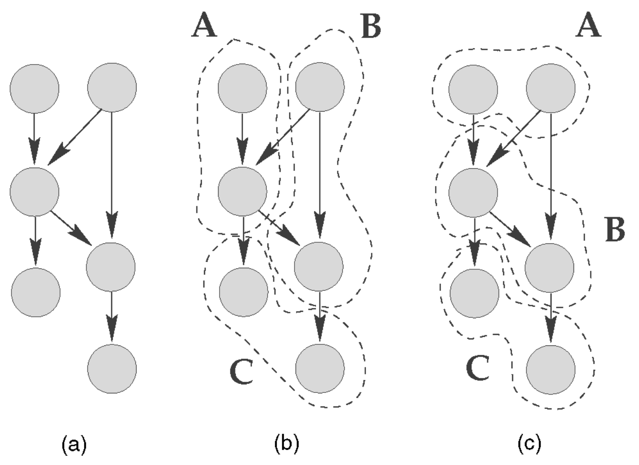


Fig. 3. Example illustrating possible formation of cycles.

**Given:** A Data Flow Graph $G = (V, E, W, D)$ as described above and a configurable unit of size $S_{RPU}$
**Objective:** Divide $G$ into $k$ segments such that:

1. size of each segment is less than or equal to $S_{RPU}$
2. there exists an acyclic `precedence-relation` for all $k$ segments.

It should be noted that just satisfying Condition 1 in the above objective can result in a cyclic[2] `precedence-relation` even when the application DFG is acyclic. One such example is illustrated in Fig. 3. In Fig. 3a, we see the unpartitioned DFG. Figs. 3b and 3c illustrate two example partitions where the size of each segment is limited to two nodes. Clearly, Fig. 3b results in a cyclic precedence relation (segments A and B) and is not feasible. On the other hand, Fig. 3c results in a feasible precedence relation.

## 5 Configurable Computer Model

In order to experimentally verify our temporal partitioning framework, we make use of a configurable computer model that is similar to the one illustrated in Fig. 1. Temporally partitioned segments are synthesized and their configurations are stored on a host computer. The configurations are dispatched to the configurable unit in the order that is determined by the precedence-relation. Temporary data buffering is accomplished using the local memory associated with the configurable computing machine.

For our experimental setup, the reconfigurable unit is part of the hardware/software co-execution environment called RACE [35]. The hardware platform, RACE-I, consists of four Xilinx XC4013 FPGAs interconnected in a complete graph ($K_4$) configuration. Each FPGA is supported with a local memory of 128 Kbytes. The RACE-I platform is connected to a SUN SparcStation using the SBUS interface. An additional XC4013 acts as a controller and the system interface. The controller is used for programming the FPGAs and DMA transfers to the host system. The memory associated with each FPGA is 8-bits wide and resides on an address and data bus that is local to that FPGA. Fig. 4

1. Size and delay are expressed in terms of the target hardware.

2. Cyclic relation will never result in *stable inputs*.

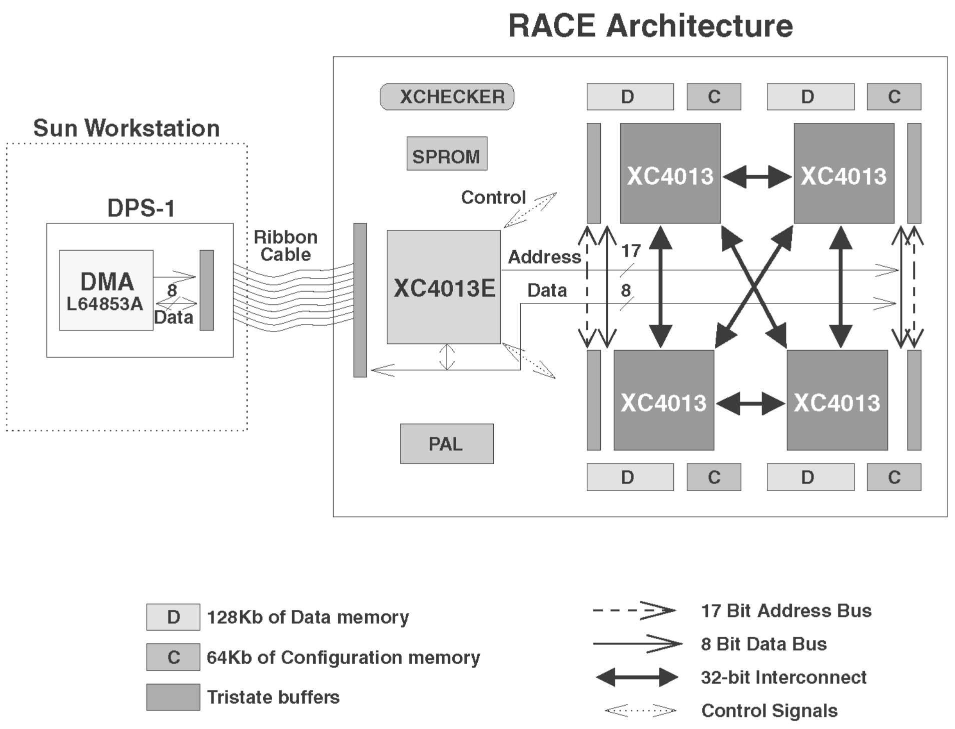


Fig. 4. Architecture of the RACE configurable computing problem.

illustrates the complete layout of the RACE-I system along with architectural details.

### 5.1 Design Considerations

To implement a design on the configurable unit, the data memory is used to provide stable inputs to the design and then store back the results. In other words, the data memory provides the necessary communication between various temporally partitioned segments. This requires the addition of a controller *finite state machine* (FSM) which performs the task of providing stable inputs to the temporal segments and writing back outputs to the data memory. We call this controller FSM the *data controller*. We have developed a synthesis tool for automatic generation of the *data controller* for the partitions (explained in Section 6.3).

## 6 DFG Partitioning

In this section, we describe various algorithms for partitioning the DFG and mapping it to configurable hardware. Fig. 5 illustrates the design flow for our DFG partitioning system. The input design specification can come from one of several commercial formats [46]. Since our processing is on DFG, G = (V, E, W, D), as described in Section 4, the input specification requires preprocessing before partitioning is carried out.

The input design specification is transformed into a graph. This allows easy identification of data dependencies between various nodes in the graph. The characteristics of the functional block are technology dependent and (in our experimental study) they are mapped to `Xilinx` XC4000 FPGAs [49]. An example graph for an application is shown in Fig. 6.

### 6.1 ASAP Level Assignment

The temporal partitioning performed under the area constraints should respect the dependencies between the nodes (to ensure correct execution). Hence, a node can be executed *iff* all its predecessors have already been executed (i.e., the inputs to every node should be stable before it is executed). For every node $v_i \in V$, we assign $Level(v_i)$, called the *ASAP level* [17], the depth of the node with respect to primary inputs. Let

- $Level(v_i)$ denotes the topological level of the node $v_i$ in the input directed acyclic graph. Initially, all the nodes of the graph are assigned a Level $= \infty$.
- *Queue* denotes the queue of nodes with Level $\neq \infty$.
- $Indegree(v_i)$ denotes the number of incoming edges for the node $v_i$ with Level $= \infty$. For all nodes $v_j \in V$ such that inputs to $v_j$ are primary inputs only, the $Indegree(v_j) = 0$.
- $FanoutSet(v_i)$ denotes the set of fanout nodes of $v_i$, i.e., $v_j \in FanoutSet(v_i)$ iff there exists a directed edge $< v_i, v_j >$ in $E$.
- $FaninSet(v_i)$ denotes the set of fanin nodes of $v_i$, i.e., $v_j \in FaninSet(v_i)$ iff there exists a directed edge $< v_j, v_i >$ in $E$.

Initially, all the nodes are assigned Level $= \infty$ and all the primary inputs of the design have $Indegree = 0$. All the primary inputs are assigned $Level = 1$ and added to the queue. Now, every node $v_i$ is removed from the queue and the Indegree of all its fanout nodes is decremented. The Indegree of a node is a measure of the number of fanin nodes with unassigned Level. If the Indegree is zero, then the node can be assigned a Level equal to one more than the maximum level of its fanin nodes, i.e., $FaninSet(v_i)$.

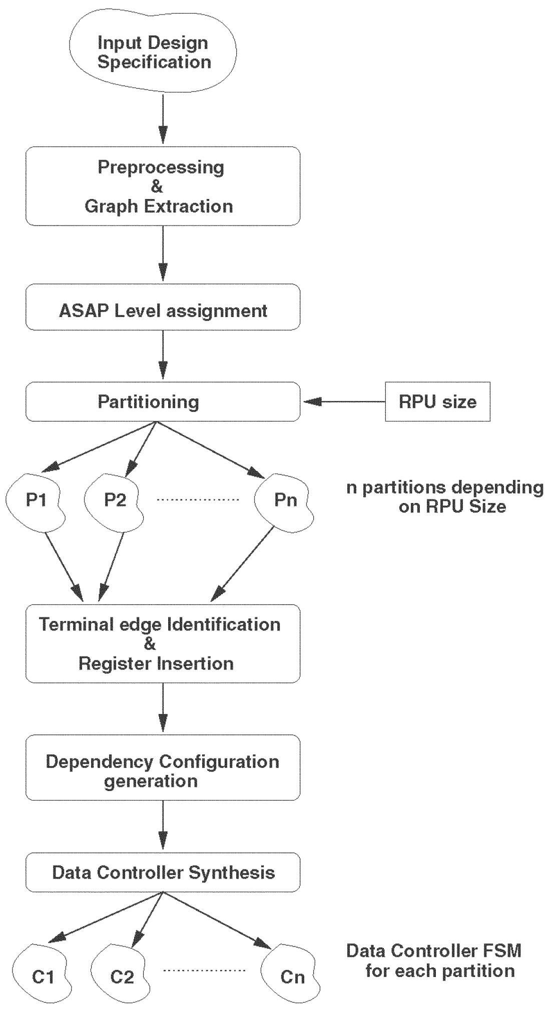

Input Design Specification
Preprocessing & Graph Extraction
ASAP Level assignment
Partitioning
RPU size
P1
P2
Pn
n partitions depending on RPU Size
Terminal edge Identification & Register Insertion
Dependency Configuration generation
Data Controller Synthesis
C1
C2
Cn
Data Controller FSM for each partition

Fig. 5. Design flow for partitioning and FSM synthesis.

$$Level(v_i) = MAX(Level(FaninSet(v_i))) + 1. \quad (1)$$

If $\mid V \mid$ denotes the size of the node set of the graph and $\mid E \mid$ denotes the size of the edge set of the graph, then the initial computation of Indegree and the identification of the primary input nodes can be done by a simple traversal of the graph (which takes about $O(\mid V \mid + \mid E \mid)$. The algorithm traverses all outgoing edges of each vertex only once, so its running time is proportional to the number of outgoing edges of the visited vertices. Since the total number of outgoing edges in the graph is equal to the size of the edge set $E$, the runtime complexity of the algorithm is $O(\mid V \mid + \mid E \mid)$.

**Algorithm: AssignLevels**

```
Queue $\Leftarrow \emptyset$
For each node $v_i \in V$ do
    If Indegree($v_i$) = 0 then
      Level($v_i$) $\Leftarrow$ 1
      Queue.Add($v_i$)
    End If
End For each
While Queue is not empty do
    For each $w_i \in$ FanoutSet($u_i$) do
      Indegree($w_i$) $\Leftarrow$ Indegree($w_i$) $-$ 1
      If Indegree($w_i$) = 0 then
        Level($w_i$) $\Leftarrow$ Level($u_i$) + 1
        Queue.Add($w_i$)
      End If
    End For each
End While
```

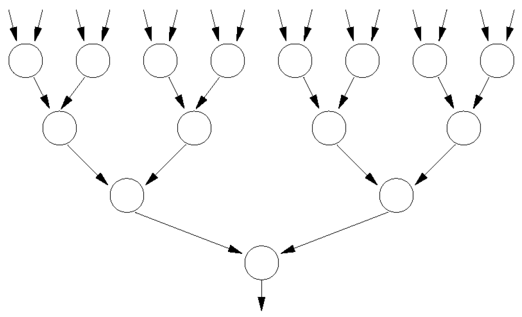


Fig. 6. An example design in node and edge format.

## 6.2 Partitioning Algorithms

The main objective of temporal partitioning is to partition the DFG under the area constraint given by the size of the configurable unit, $S_{RPU}$. Additionally, a second equally important objective is to exploit the inherent parallelism that is present in an application. (This second objective is typically used in partitioning graphs for multiprocessors [33].) However, when the degree of parallelism exploited increases, a new performance bottleneck arises in the form of communication overhead for satisfying the data dependencies between the partitions. Hence, it is a trade-off to come up with a method that extracts maximum performance from the available resources.

To quantitatively study the above trade-off, we have designed two different partitioning algorithms. A *level-based partitioning* algorithm and a *clustering-based partitioning* algorithm. The first algorithm tries to achieve maximum possible parallelism, thereby decreasing the delay. The second algorithm tries to minimize the communication overhead by sacrificing the parallelism, i.e., increasing the delay.

We study the trade-off between decreasing the delay of the partition (by making use of parallelism) to the communication overheads associated with such parallelism. Our study involves the execution of the temporal partitions on the configurable unit model explained in Section 5. Hence, the communication overhead in our case is the size of the *data controller* that addresses the data communication between the sequentially executing segments. We also substantiate our study with various example designs and illustrate the overheads.

### 6.2.1 Level Based Partitioning

The level assignment algorithm classifies the nodes of the input application according to their ASAP levels [17]. The ASAP levels also expose the parallelism hidden in the graph nodes, i.e., all the nodes with the same level can be considered for parallel execution without any dependency check. There also exists some degree of parallelism among the nodes with different levels (if they are not connected by an edge).

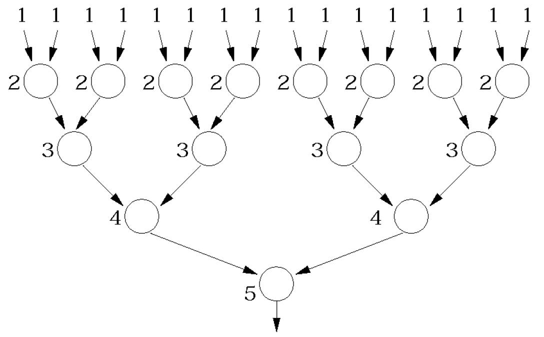


Fig. 7. ASAP Level assignment for the example design.

All the primary inputs to the design are assumed to have a size of zero. A conservative approach would be to execute nodes in increasing order relative to their ASAP levels; this ensures stable inputs for every node at the next level. Such an approach would also exploit the maximum parallelism from the input application and preserve its acyclic properties.

We have characterized the overheads involved in executing logic partitions on our configurable computer model. These overheads are mainly due to the limited availability of routing resources and the necessity for a *data controller* to perform the data transfer across the temporal partitions. Thus, for a given configurable computer,

$$Available\_Area = S_{RPU} - FSMCost - RCost. \tag{2}$$

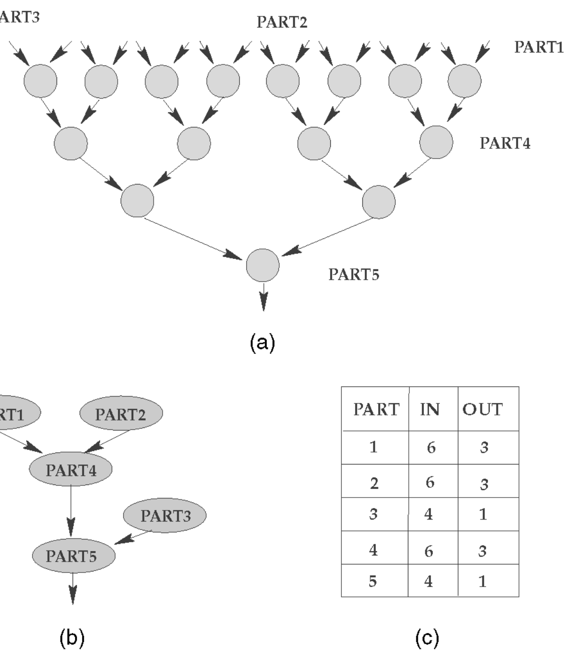
*FSMCost* is the cost (in terms of the number of configurable logic blocks) for the *data controller* logic. *RCost* is a constant (characterized for an FPGA architecture) that ensures the routability of the logic on the FPGA. *RCost* can be expressed in terms of the percentage of the logic residing on the configurable unit.

Our experimental studies (see Section 6.3) have identified that the size of such an FSM is proportional to the communication overhead, which depends on the number of *terminal edges* of a partition. Terminal edges for a partition are the collection of the incoming edges and the outgoing edges for the given partition. We have characterized the *FSMCost* for a wide range of terminal edges (see Section 6.3) and incorporated it in our cost model. $Partition(v_i)$ denotes the partition to which the node $v_i, 1 \le i \le N$, belongs. $Max\_Level$ denotes the maximum level of any node in the graph.

**Algorithm II: Level-based partitioning algorithm**

```
For each node $v_i$ with Level($v_i$) = 0 do
  Partition($v_i$) ⇐ 0
End For Each
$i \Leftarrow 2$
Lev ⇐ 2
Area_Filled ⇐ 0

While(Lev ≤ Max_Level)
  For each node $v_i$ with $Level(v_i)$ = Lev do
    e ⇐ Identify_Terminal_Edges($v_i$)
    Total_Cost ⇐ Calculate_FSMCost(e) + Size($v_i$) + RCost
    If((Area_Filled + Total_Cost ≤ $S_{RPU}$) then
      Partition($v_i$) ⇐ $i$
      Area_Filled ⇐ Area_Filled + Total_Cost
    End If
    Else
      $i \Leftarrow i + 1$
      Partition(v_i) ⇐ i
      Area_Filled ⇐ Total_Cost
    End Else
  End For each
  Lev ⇐ Lev + 1
End while
```



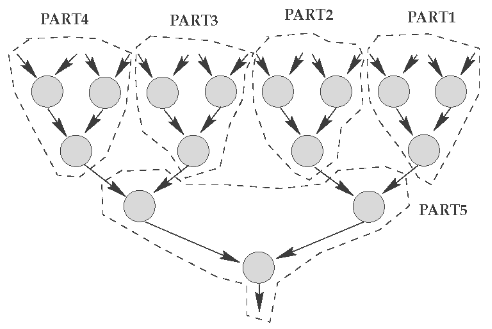
| PART | IN | OUT |
|---|---|---|
| 1 | 6 | 3 |
| 2 | 6 | 3 |
| 3 | 4 | 1 |
| 4 | 6 | 3 |
| 5 | 4 | 1 |

Fig. 8. Level-based partitioning of the example design.

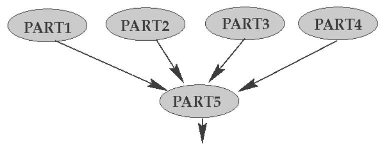
The algorithm traverses each node of the graph, level by level, and assigns them to a partition. All primary inputs to the design are assigned to the first partition because of their zero size. The remaining nodes are assigned to partitions numbered 2 and beyond. All the nodes from level 2 to Max_Level are traversed. Nodes of the same level are packed in a single partition and if the available area is exhausted, then the nodes are assigned to the next partition. If the nodes in the current level are all assigned to a partition, then the next level nodes are considered. To start with, a partition has no nodes and no terminal edges. Before adding a node, the additional costs associated with the change in the terminal edges is calculated using Identify_Terminal_Edges().

When a node is added to a partition, also called as segment, the size of terminal edges will also change which will have direct impact on the size of the data controller FSM. In order to satisfy the area constraints, we need to correctly estimate the size of the segment. Let $X$ be the number of terminal edges for the segment prior to adding a node $v_i$, $1 \le i \le N$, $Y$ be the number of fanin edges to $v_i$ that originate from nodes already in the current segment, and $Z$ be the fanout edges from $v_i$. Then, the change, $C$, in the number of terminal edges due to the addition of $v_i$ to the segment:

$$C = Z - Y. \tag{3}$$



(a)



| PART | IN | OUT |
|---|---|---|
| 1 | 4 | 1 |
| 2 | 4 | 1 |
| 3 | 4 | 1 |
| 4 | 4 | 1 |
| 5 | 4 | 1 |

(b) (c)

Fig. 9. Clustering based schedule for the example design.

The total number of terminal edges upon addition of $v_i$ to the segment is given by $X + C$. Fig. 8a illustrates the partitions generated from the level based partitioning algorithm on an example design. Fig. 8b illustrates the resulting graph evolving from level based partitioning on the example design. Fig. 8c illustrates the total number of incoming and outgoing terminal edges resulting from level based partitioning.

If $\mid V \mid$ denotes the size of the node set of the graph and $\mid E \mid$ denotes the size of the edge set of the graph, then the assignment of primary inputs to the first partition takes constant time (as the nodes are arranged in a level order). The nodes are traversed from level 2 to Max_Level and, for every node, the change in the terminal edges in calculated. This calculation involves checking of the fanin and fanout edges of a node. For all nodes, since fanin and fanout are checked twice (once for the node itself and the second time while evaluating the node(s) connected to this node), it has a complexity of $2 \mid E \mid$. The calculation of the FSMCost and RCost is a constant time operation performed for all nodes (i.e., $\forall v_i \in\mid V \mid$). Hence, the total complexity of the algorithm is $O(\mid V \mid + \mid E \mid)$.

### 6.2.2 Clustering Based Partitioning Algorithm

In order to decrease the communication overhead, i.e., the number of terminal edges resulting from partitioning, we have developed an algorithm that clusters nodes with common parent. This has a tendency to assign the common parent to the same partition, provided area is available, as rest of the children nodes, thereby decreasing the number of terminal edges.

**Algorithm III: Clustering based partitioning**

```
ReadyList ⇐ ∅
For each node $v_i \in V$
    Calculate_successor_id($v_i$)
End For each
For each node $v_i \in V$ with Level($v_i$) = 1
    Partition($v_i$) ⇐ 1
    ReadyList.update($v_i$)
End For each

$i \Leftarrow 2$
While ReadyList is not empty do
    $u_i$ ⇐ ReadyList.front()
    e ⇐ Identify_Terminal_Edges($u_i$)
    Total_Cost ⇐ Calculate_FSMCost(e) + Size($u_i$) + RCost
    If((Area_Filled + Total_Cost ≤ $S_{RPU}$) then
        Partition($u_i$) ⇐ $i$
        Area_Filled ⇐ Area_Filled + Total_Cost
    End If
    Else
        $i \Leftarrow i + 1$
        Partition($u_i$) ⇐ $i$
        Area_Filled ⇐ Total_Cost
    End Else
    ReadyList.update($u_i$)
End While
```

To speed up the search (for nodes with common parents at a particular level), we store the `id` of the successor node with each node. In case of nodes with more than one fanin, the `id` of the node closer to this node is stored. Whenever a node is assigned to a partition, the possibility of assigning any of its successor nodes to the current partition is also

```
PARTITION, 4
INPUT, 1, PARTITION, 1, EDGE, 1, BYTES, 1
INPUT, 2, PARTITION, 1, EDGE, 4, BYTES, 1
INPUT, 3, PARTITION, 3, EDGE, 1, BYTES, 1
INPUT, 4, PARTITION, 1, EDGE, 5, BYTES, 1
INPUT, 5, PARTITION, 2, EDGE, 3, BYTES, 1
TOTAL_INPUT, 5
OUTPUT, 1, EDGE, 1, BYTES, 1, PRIMARY_OUTPUT, 1
TOTAL_OUTPUT, 1
DELAY, 4
END
```

Fig. 10. An example dependency configuration file.

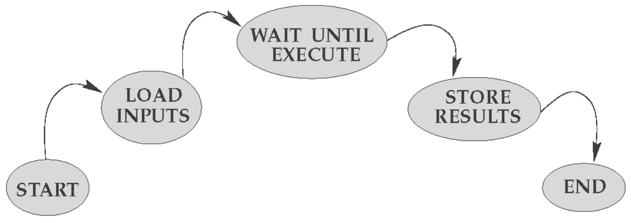


Fig. 11. Data controller embedded in each segment ensures proper execution.

checked. Because of the heuristics applied to improve the speed of the algorithm, this may not result in an optimal solution, but our experimental results have proved a substantial improvement in the communication overhead.

In the algorithm, *ReadyList* represents a queue with the nodes that are ready to be executed(i.e., all of the nodes in its *FaninSet* have already been scheduled in the current or previous segments). Initially, it consists of only primary inputs. All primary inputs are assigned to partition 1. *ReadyList.update()* adds new nodes that are ready to execute at the front of ReadyList. The new nodes joining the list are given more priority by adding them at the front of the list. This has the potential to decrease the terminal edges. This algorithm ensures better results in communication overhead when compared to the level-based scheme.

The application of the algorithm on the example design is illustrated in Figs. 8b and 9b. These figures illustrate the newly formed dependencies in terms of the terminal edges between the partitions. The corresponding number of terminal edges for every segment is shown in Figs. 8c and 9c. In the figures, **IN** represents the set of incoming terminal edges of a partition and **OUT** represents the set of outgoing terminal edges.

The calculation of successor `id` involves identifying the fanout nodes for every node, which is proportional to the number of edges, i.e., $|E|$. The update function checks if any of the fanout nodes are ready to assign to a partition. This is done for every node in the design, once for all the primary inputs and then for the remaining nodes. This also

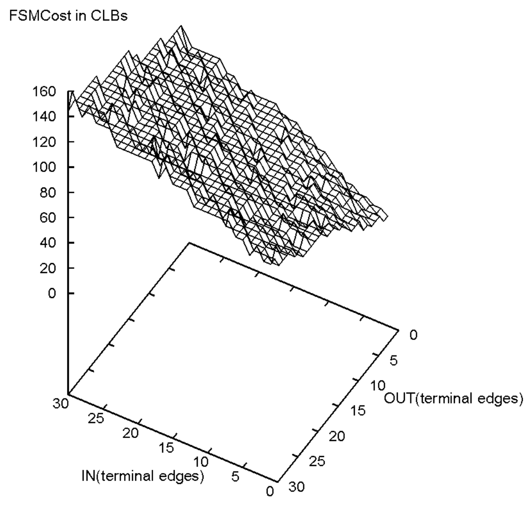


Fig. 12. Variation of FSM size with terminal edges.

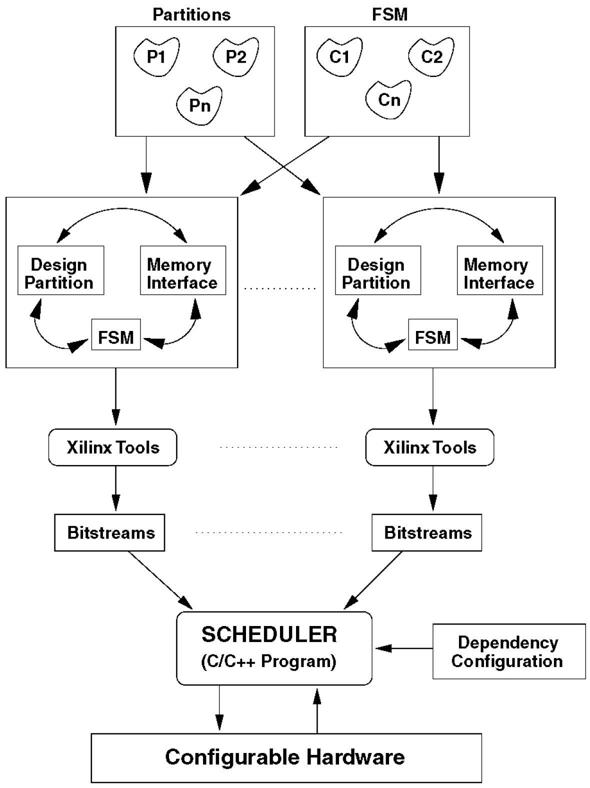


Fig. 13. Scheduling data flow graphs.

requires $O(|E|)$ time. The computational time involved in identifying terminal edges is $2|E|$ and the computational time for calculating FSMCost and RCost is $O(|V|)$. Hence, the total computational complexity of the algorithm is $O(|V| + |E|)$.

The data dependencies between the temporal partitions should be stored in a data format. During the runtime, the *Scheduler* handles the data dependencies with the help of this data and ensures correct execution.

An example configuration file is shown in Fig. 10. The out-bound terminal edges of a segment are assigned with a unique order number. The in-bound terminal edge of any

```
RPU.Load(partition4.bit)
IN[4][1][1] = OUT[1][1][1]
IN[4][2][1] = OUT[1][4][1]
IN[4][3][1] = OUT[3][1][1]
IN[4][4][1] = OUT[1][5][1]
IN[4][5][1] = OUT[2][2][1]
RPU.LoadMemory(IN[4])
RPU.start()
While RPU.done() ≠ true
    RPU.wait()
End While
RPU.StoreMemory(OUT[4])
```

Fig. 14. Sample code for executing a partition on the hardware.

TABLE 1
Size of the Benchmark Circuits Expressed in XC4000 CLBs

| Benchmark | No.of nodes | Size |
|---|---|---|
| MEDIAN | 19 | 475 |
| BTREE32 | 31 | 775 |
| DCT8 | 90 | 4000 |
| MATRIX4 | 112 | 6512 |

segment can be associated with an out-bound terminal edge of any of the previous segments. The output data from each segment is stored according to the unique order number and retrieved when required by the successive segments.

### 6.3 Data Controller Synthesis

In Section 6.2.1, we introduced a new cost, FSMCost, that should be considered while partitioning. The FSMCost represents the logic utilized for the communication overhead. We synthesize the FSMs automatically [30] from the size of IN and OUT and the estimated delay for a segment. We generate ABEL [47] files representing the data controller functionality. The functionality of the data controller is represented as shown in Fig. 11. It loads the input from memory to the IN edges and then waits until the end of the execution. Finally, the FSM loads the results, OUT, into the memory.

We have characterized the FSMCost as a function of the terminal edges, i.e, IN and OUT, as shown in Fig. 12. To calculate the FSMCost overhead while assigning a node to a partition, the total size of IN and OUT is calculated and referred to the cost table to find out the FSMCost.

## 7 DFG Scheduling

Scheduling of the temporal partitions onto the configurable hardware should satisfy:

1. Precedence relation between the partitions.
2. Data dependencies among the partitions.

The partitions generated from the algorithms explained in Sections 6.2.1 and 6.2.2 have a uniform interface. A set of in-bound terminal edges and a set of out-bound terminal edges are supported by a *data controller* to perform the data transfer between logic and memory. The hardware mappings are generated using vendor specific synthesis and mapping tools. Fig. 13 illustrates the flow of events occuring in scheduling the segments.

We have developed a data structure to manage the data dependencies between the partitions. The scheduler is a software program invoking the driver functions to load configuration memory and data memory on the hardware. The data structure consists of two three-dimensional arrays:

```
IN[p][i][w] and OUT[p][o][w]
p - Partition number
i - Incoming terminal edge number
o - Outgoing terminal edge number
w - Width of the terminal edge, expressed in
    bytes.
```

While executing a partition `k`, the scheduler loads all `IN[k][i][w]` and waits until the end of execution to read `OUT[k][o][w]` from the data memory. The inputs, `IN[k][i][w]`, are assigned from outputs of the previous partitions (i.e., `OUT[j][o][w]`, where $j < k$). To execute the partition shown in the Fig. 10, the corresponding scheduler code is shown in Fig. 14.

## 8 Results and Analysis

In order to experimentally verify the concept of temporal partitioning and scheduling of the applications, we have designed four benchmarks. *MEDIAN* is a median filter used for reducing shot noise in images [36]. *BTREE32* is a 32 number binary tree comparator. It compares 32, 8-bit numbers in parallel and outputs the largest of the inputs. *MATRIX4* is a $4 \times 4$ integer, matrix multiplier completely implemented in parallel. *DCT8* [38], [10] is a one-dimensional discrete cosine transform, implemented completely in parallel. DCT-based image coding is the basis for all image and video compression standards [45]. DCT transforms an $N \times N$ image block from the spatial domain to the DCT domain. One-dimensional DCT can be used to compute a two-dimensional DCT. The number of nodes and the approximate sizes of these circuits are shown in Table 1.

The benchmark circuits are executed both on the software and hardware and their execution times are compared. The software execution of the benchmarks is carried out on three different machine configurations and the execution times are reported in Table 2. In order to

TABLE 2
Software Execution Times on Various Class of Workstations

| Application | Sparc-5 time (msec) | Ultrasparc-1 time (msec) | Ultrasparc-2 time (msec) |
|---|---|---|---|
| MEDIAN | 11.53 | 6.85 | 3.26 |
| BTREE32 | 11.55 | 7.74 | 3.27 |
| DCT8 | 11.702 | 9.28 | 3.29 |
| MATRIX4 | 11.638 | 9.30 | 3.31 |

TABLE 3
Execution Times on Reconfigurable Hardware

| Application | Size of partition | Number of partitions($k$) | Hardware Execution time $T_{hardwareexecution}(\mu sec)$ |
|---|---|---|---|
| MEDIAN | 200 | 2 | 33.6 |
| BTREE32 | 450 | 2 | 16.68 |
| DCT8 | 450 | 9 | 87.94 |
| MATRIX4 | 450 | 16 | 53.8 |

accurately measure the execution time in software, we make use of the *quantify* [31] utility which does execution profiling and reports real processor cycles spent on executing a task and its descendants.

The execution times of the benchmark circuits on reconfigurable hardware with the temporal partitioning approach is shown in Table 3. The table describes the number of partitions made for each application and also the size of each partition. For the tabulated results, the *clustering-based scheduling* algorithm is used. Note that there is a reconfiguration overhead of 242 msec for each segment configuration. Thus, the total time for executing a benchmark would be

$$T_{exec} = (k * 242msec) + T_{hardwareexecution}, \tag{4}$$

where $k$ represents the number of temporal partitions and $T_{hardwareexecution}$ represents the total time taken to execute all the temporal partitions of a computation on the hardware.

The reconfiguration overhead is dependent on the architecture of the reconfigurable computer. The experimental setup RACE [34] consists of an overhead of 242 msec. The recent architectures [7], [12] have scaled down the overhead by at least 1,000 times. We later explain a methodology to optimize the reconfiguration overheads for data intensive computations.

## 8.1 Comparison of the Partitioning Algorithms

Fig. 15 indicates an improvement of at least 30 percent in terminal edges for the example designs using the clustering based scheduling algorithm (for $S_{RPU} =$ 576 CLBs). The number of terminal edges reported in Fig. 15 are averaged over all the partitions of the application. Fig. 16 indicates at least 67 percent degradation in the average delay of execution for a partition from level-based algorithm to clustering-based algorithm.

We chose the parameters *terminal edges* and *delay* to quantitatively evaluate the partitioning algorithms. Though these parameters seem to be target configurable unit model dependent, analogous performance can be identified in parallel and multiprocessing research areas [33]. The *terminal edges* parameter is analogous to the communication overhead between parallel executing processes in a multiprocessing environment. The *delay* is analogous to the execution delay of each process in a multiprocessing environment. The partitioning algorithms proposed in the above sections can be extended to the parallel processing domain with a slight change in the performance parameters and cost function. Hence, the partitioning strategy to be adopted is subjective to the user constraints and the target architecture of parallel computer or a configurable computer.

Fig. 17 shows the key functional blocks in a generic DCT based coding system [45]. The computation pipeline makes

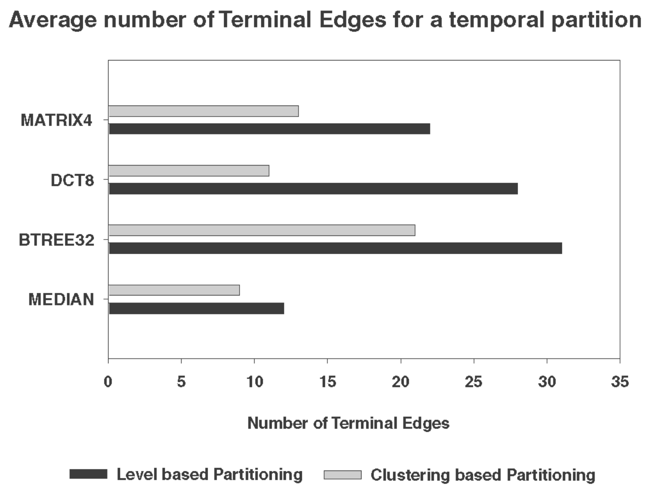


Fig. 15. Performance analysis of the partitioning algorithms in terminal edges.

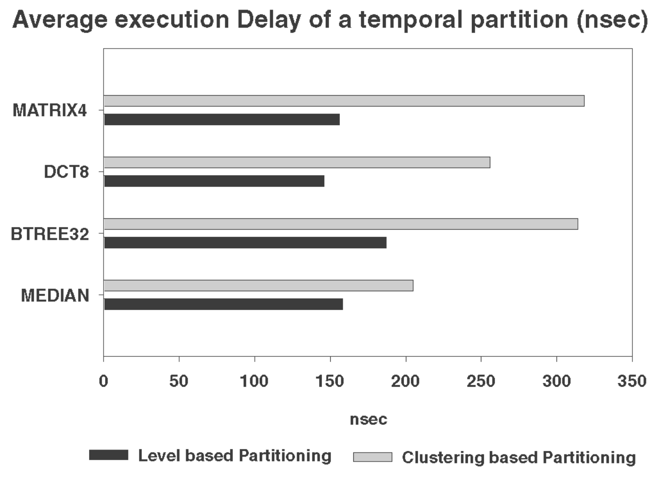


Fig. 16. Performance analysis of the partitioning algorithms in delay.

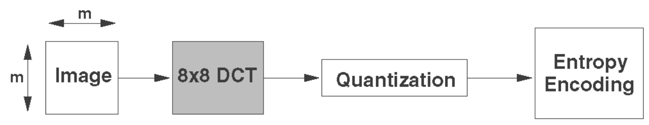


Fig. 17. Discrete cosine transform.

use of DCT as first block for most lossy image compression algorithms. Our comparison highlights the first block (shaded region) of Fig. 17. For software based execution, the total time for execution is given by

$$T_{soft-DCT} = \left\lceil \frac{m}{8} \right\rceil \times \left\lceil \frac{m}{8} \right\rceil \times T_{soft-DCT8}, \qquad (5)$$

where $T_{soft-DCT}$ is the time for image transformation to DCT domain for an $m \times m$ image when executed in software and $T_{soft-DCT8}$ is the time for executing an $8 \times 8$ DCT in software.

In order to gain most by hardware execution, we try to minimize the need for reconfiguration. If $k$ is the number of temporal segments, then we require $k$ reconfigurations only if the $i$th, $1 \leq i \leq k-1$, temporal segment executes completely on entire $m \times m$ image and stores temporary results before the configurable hardware is reconfigured with the $(i+1)$th segment. Thus, hardware-based execution results in

$$T_{hard-DCT} = \sum_{i=1}^{k} \left\lceil \frac{m}{8} \right\rceil \times \left\lceil \frac{m}{8} \right\rceil \times T^{i}_{hard-DCT8} + k \cdot T_{reconfig}, \qquad (6)$$

where $T_{hard-DCT}$ is the time for image transformation using configurable hardware; $T^{i}_{hard-DCT8}$ is the time for executing $i$th, $1 \leq i \leq k$, segment for $8 \times 8$ DCT in hardware; and $T_{reconfig}$ is the reconfiguration time. Clearly, for $m$ as small as 128, the hardware performance will exceed the software performance. However, in our case, we can represent an $8 \times 8$, 2D-DCT by 16 executions of 1DCT8. Thus, for very small values of $m$, we will see reconfiguration overheads completely absorbed and, for large images, the performance of our system will far exceed the software execution of DCT transformations.

## 9 Conclusions

In this paper, we have presented a novel approach to partition and schedule DFGs in the time domain. The aim of this approach is to implement very large applications on small reconfigurable hardware. We have developed a complete hardware environment and necessary CAD tools required for processing and synthesizing large applications. This work also proposes an intermediate format and a communication protocol for sequentially executing partitions in the time domain. We have also discussed the overheads involved in the current state-of-the-art configurable logic devices and their role on the performance bottlenecks through quantitative results.

This approach has wide applications in hardware logic emulation and algorithm acceleration using configurable logic. The approach to temporally partition and schedule the application reuses the available hardware, thereby cutting down the emulation costs as well as making the emulation system capable of handling large applications without scaling (i.e., adding new hardware [29]). It also tries to simplify the complex task of spatial partitioning for multi-FPGA boards by implementing linear time algorithms for temporal partitioning. Our approach helps make a configurable computing board scalable by appropriately generating the temporal mappings and reusing the hardware. The potential gains are in the hardware costs and speed. Also, for new and evolving architectures like context switch-able FPGAs, i.e., FPGAs with more than one configuration memory bank and an ability to switch configuration from one to another in almost single clock cycle, can make use of our methodology for effective execution of large applications with no reconfiguration overhead.

With advances in the FPGA technology toward a million programmable logic gates, programmable logic with embedded memories [50] and the coupling of configurable logic on the processor core, this approach provides good applicability for efficient reuse of the hardware and thus scalability of the available resources. It also opens some interesting problems that are worthy of further investigation. For example, as the granularity of the programmable hardware ($S_{RPU}$) increases, an application may not require any temporal partitioning after certain level of device logic density. The trade-off between performance and cost for changing device granularity would be an interesting relationship that is worthy of further study. In other words, what is the minimum granularity that is needed to achieve performance that far exceeds the performance that is obtainable on a general purpose machine.

## Acknowledgments

This research is partially suported by the U.S. Air Force Laboratory at Wright Patterson Air Force Base under contract number F33615-96-C1912.

## References

[1] A.V. Aho, R. Sethi, and J.D. Ullman, *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1986.

[2] O.T. Albahama, P. Cheung, and T.J. Clarke, "On the Viability of FPGA-Based Integrated Coprocessors," *Proc. IEEE Symp. FPGAs for Custom Computing Machines,* K.L. Pocek and J. Arnold, eds., pp. 206-215, Apr. 1996.

[3] P.M. Athanas and A.L. Abbott, "High-Speed Image Processing with Splash 2," *Splash2, FPGAs in Custom Computing Machine,* pp. 141-165. IEEE CS Press, 1996.

[4] J. Babb, R. Tessier, M.D. Silvina, Z. Hanono, D.M. Hoki, and A. Agarwal, "Logic Emulation with Virtual Wires," *IEEE Trans. Computer-Aided Design of Integrated circuits and Systems,* vol. 16, no. 6, pp. 609-626, June 1997.

[5] N.B. Bhat, "Novel Techniques for High Performance Field Programmable Logic Devices," Technical Report ERL-93-80, Computer Science Division, Univ. of California, Berkeley, Nov. 1993.

[6] N.B. Bhat, K. Chaudhary, and E.S. Kuh, "Performance Oriented Fully Routable Architecture for a Field Programmable Logic Device," Technical Report ERL-93-42, Computer Science Division, Univ. of California, Berkeley, June 1993.

[7] D. Bhatia, P. Kannan, K. Simha, and K.M. Gajjala Purna, "REACT: Reactive Environment for Reconfigurable Computing," *Field-Programmable Logic: Smart Applications, New Paradigms and Compilers,* R.W. Hartenstein and A. Keevallik, eds. Berlin: Springer-Verlag, Aug./Sept. 1998.

[8] S. Brown and J. Rose, "FPGA and CPLD Architectures: A Tutorial," *IEEE Design and Test of Computers,* vol. 13, no. 2, pp. 42-57, Summer 1996.

[9] D.A. Buell, J.M. Arnold, and W.J. Kleinfelde, *Splash2, FPGAs in Custom Computing Machine.* IEEE CS Press, 1996.

[10] W.-H. Chen, C.H. Smith, and S.C. Fralick, "A Fast Computational Algorithm for the Discrete Cosine Transform," *IEEE Trans. Comm.,* vol. 25, no. 9, pp. 1,004-1,009, Sept. 1977.

[11] Gatefield Corporation, www. gatefield. com.

[12] Xilinx Corporation "The Virtex Series of FPGAs 1,000,000 System Gates at 100+ mhz,"http://www.xilinx.com/products/virtex.htm.

[13] D.A. Patterson and J.L. Hennessy, *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 1996.

[14] A. DeHon, "DPGA-Coupled Microprocessors: Commodity ICs for the Early 21st Century," *Proc. IEEE Workshop FPGAs for Custom Computing Machines,* D.A. Buell and K.L. Pocek, eds., pp. 31-39, Napa, Calif., Apr. 1994.

[15] A. DeHon, "DPGA Utilization and Application," *Proc. ACM/SIGDA Int'l Symp. Field Programmable Gate Arrays,* pp. 115-121, Monterey, Calif., Feb. 1996.

[16] M.J. Flynn, *Computer Architecture: Pipelined and Parallel Processor Design.* Jones and Bartlett Publishers, 1995.

[17] D.D. Gajski, N.D. Dutt, A.C.-H. Wu, and S.Y.-L. Lin, *High-Level Synthesis, Introduction to Chip and System Design.* Kluwer Academic, 1992.

[18] S. Gehring and S.H.-M. Ludwig, "The Trianus System and Its Application to Custom Computing," *Field-Programmable Logic: Smart Applications, New Paradigms and Compilers,* pp. 176-184. Berlin: Springer-Verlag, Sept. 1996.

[19] J.P. Gray and T.A. Kean, "Configurable Hardware: A New Paradigm for Computation," *Proc. Decennial CalTech Conf. VLSI,* pp. 277-293, Pasadena, Calif., Mar. 1989.

[20] S. Hauck, "Multi-FPGA Systems," PhD thesis, Univ. of Washington, 1997.

[21] S. Hauck, "The Role of FPGAs in Reprogrammable Systems," *Proc. IEEE,* vol. 86, no. 4, pp. 615-638, Apr. 1998.

[22] D.T. Hoang, "Searching Genetic Databases on Splash 2," *Proc. IEEE Workshop FPGAs for Custom Computing Machines,* D.A. Buell and K.L. Pocek, eds., pp. 185-191, Apr. 1993.

[23] J. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. Touati, and P. Boucard, "Programmable Active Memories : Reconfigurable Systems Come of Age," *IEEE Trans. VLSI,* vol. 4, no. 1, Mar. 1996.

[24] N.K. Ratha, A.K. Jain, and D.T. Rower, "Fingerprint Matching on Splash2," *Splash2, FPGAs in Custom Computing Machine,* pp. 117-140, 1996.

[25] M.C. McFarland, A.C. Parker, and R. Camposano, "The High-Level Synthesis of Digital Systems," *Proc. IEEE,* pp. 301-318, Feb. 1990.

[26] P.G. Paulin and J.P. Knight, "Force Directed Scheduling for the Behavioural Synthesis of ASICs," *IEEE Trans. Computer-Aided Design,* vol. 8, June 1989.

[27] D.V. Pryor, M.R. Thistle, and N. Shirazi, "Text Searching on Splash 2," *Proc. IEEE Workshop FPGAs for Custom Computing Machines,* D.A. Buell and K.L. Pocek, eds., pp. 172-177, Apr. 1993.

[28] K.M. Gajjala Purna, "Temporal Partitioning and Scheduling Data Flow Graphs for Reconfigurable Computers," Master's thesis, Dept. of Electrical and Computer Eng. and Computer Science, Univ. of Cincinnati, Oct. 1998.

[29] K.M. Gajjala Purna and D. Bhatia, "Emulating Large Designs on Small Reconfigurable Hardware," *Proc. Ninth IEEE Int'l Workshop Rapid System Prototyping,* June 1998.

[30] K.M. Gajjala Purna and D. Bhatia, "Temporal Partitioning and Scheduling for Reconfigurable Computing," Technical Report TR 212/01/98/ECECS, Dept. of ECECS, Univ. of Cincinnati, Jan. 1998.

[31] Rational Software Corporation, *Quantify User's Guide, Version 3.1,* 1997.

[32] R. Jain, A. Majumdar, A. Sharma, and H. Wang, "Empirical Evaluation of Some High-Level Synthesis Scheduling Heuristics," *Proc. 28th ACM/IEEE Design Automation Conf.,* pp. 210-215, June 1991.

[33] V. Sarkar, *Partitioning and Scheduling Parallel Programs for Multiprocessors.* MIT Press, 1989.

[34] D. Smith, "RACE : A Reconfigurable and Adaptive Computing Environment," Master's thesis, Dept. of ECECS, Univ. of Cincinnati, June 1997.

[35] D. Smith and D. Bhatia, "RACE: Reconfigurable and Adaptive Computing Environment," *Field-Programmable Logic: Smart Applications, New Paradigms and Compilers,* pp. 87-95. Berlin: Springer-Verlag, Sept. 1996.

[36] J.L. Smith, "Implementing Median Filters in XC4000 FPGAs," *Xcell Articles.* Xilinx Corporation, San Jose, Calif., Quarter 4, 1996.

[37] W. Stallings, *Computer Organization and Architecture: Designing for Performance.* Prentice Hall, 1996.

[38] B. Stott, D. Johnson, and V. Akella, "Asynchronous 2-D Discrete Cosine Transform Core Processor," *Proc. Int'l Conf. Computer Design, ICCD95,* Oct. 1995.

[39] Ikos Systems, www. ikos. com.

[40] Quickturn Design Systems, www. quickturn. com.

[41] S. Trimberger, "Scheduling Designs into a Time-Multiplexed FPGA," *Proc. ACM/SIGDA Int'l Symp. Field Programmable Gate Arrays, FPGA98,* pp. 153-160, Feb. 1998.

[42] S. Trimberger, D. Carberry, A. Johnson, and J. Wong, "A Time-Multiplexed FPGA," *Proc. IEEE Workshop FPGAs for Custom Computing Machines,* pp. 22-28, Apr. 1997.

[43] TSI-Telsys Inc., *ACE Card, User's Manual, Version 1. 0,* 1998.

[44] J. Varghese, M. Butts, and J. Batcheller, "An Efficient Logic Emulation System," *IEEE Trans. VLSI Systems,* vol. 1, no. 2, pp. 171-174, June 1993.

[45] V. Bhaskaran and K. Konstantinides, *Image and Video Compression Standards.* Kluwer Academic, 1997.

[46] Xilinx Corporation, *Xilinx Netlist Format (XNF) Specification, Version 6. 1,* San Jose, Calif., 1995.

[47] Xilinx Corporation, *Xilinx XABEL Reference Manual,* San Jose, Calif., 1995.

[48] Xilinx Corporation, *Gate Count Capacity Metrics for FPGAs, XAPP 059,* San Jose, Calif., 1997.

[49] Xilinx Corporation, *The Programmable Logic Data Book,* San Jose, Calif., 1998.

[50] Xilinx Corporation, *Virtex 2. 5v FPGA Series(XCv00),* San Jose, Calif., Oct. 1998.

**Karthikeya M. Gajjala Purna** received his BTech from the Regional Engineering College, Warangal (1996) and his MS from the University of Cincinnati (1998). He is currently working with Synopsys Inc. in Sunnyvale, California. His research interests include design and development of CAD tools, synthesis, and simulation of digital systems. He is a member of the ACM and the IEEE Computer Society.

**Dinesh Bhatia** is an associate professor in the Department of Electrical and Computer Engineering and Computer Science at the University of Cincinnati. He also directs the Design Automation Laboratory. His research interests include all aspects of architecture and CAD for field programmable gate arrays, reconfigurable and adaptive computing, physical design automation of VLSI systems, and algorithms. His research is actively supported by the U.S. Defense Advanced Research Projects Agency (DARPA) and the United States Air Force. He is a member of the IEEE, the ACM, and Eta Kappa Nu.