On Combining Temporal Partitioning and Sharing of Functional Units in Compilation for Reconfigurable Architectures*

João M. P. Cardoso

Faculty of Sciences and Technology/University of Algarve

Campus de Gambelas

8000 – 117 Faro

Portugal

Phone: +351 936064195

Fax: +351 289819403

Email: jmpc@acm.org

Also with INESC-ID, Lisboa

Submitted for a Regular Paper

*** this paper is an improved version of the preliminary following paper:**

João M. P. Cardoso, **"A Novel Algorithm Combining Temporal Partitioning and Sharing of Functional Units,"** In *IEEE 9th Symposium on Field-Programmable Custom Computing Machines (FCCM'01)*, Rohnert Park, California, USA, April 30 – May 2, 2001, IEEE Computer Society Press, Los Alamitos, CA, USA (*to appear*).

On Combining Temporal Partitioning and Sharing of Functional Units in Compilation for Reconfigurable Architectures

João M. P. Cardoso

Faculty of Sciences and Technology/University of Algarve

Campus de Gambelas

8000 – 117 Faro

Portugal

Phone: +351 936064195

Fax: +351 289 819403

Email: jmpc@acm.org

Also with INESC-ID, Lisboa

Abstract

Resource virtualization on FPGA devices, achievable due to its dynamic reconfiguration capabilities, provides an attractive solution to save silicon area. Architectural synthesis for dynamically reconfigurable FPGA-based digital systems needs to consider the case of reducing the number of temporal partitions (reconfigurations), by enabling sharing of some functional units in the same temporal partition. This paper proposes a novel algorithm for automated datapath design, from behavioral input descriptions (represented by a dataflow graph), which simultaneously performs temporal partitioning and sharing of functional units. The proposed algorithm attempts to minimize both the number of temporal partitions and the execution latency of the generated solution. Temporal partitioning, resource sharing, scheduling, and a simple form of allocation and binding are all integrated in a single task. The algorithm is based on heuristics and on a new concept of construction by gradually enlarging timing slots. Results show the efficiency and effectiveness of the algorithm when compared to existent approaches.

Index Terms: FPGA, Reconfigurable Computing, Scheduling, Temporal Partitioning

1 Introduction

The availability of multiprogrammable logic devices (such is the case of FPGAs - field programmable gate arrays) with lower reconfiguration times has made possible the concept of “virtual hardware” [1][2]: the hardware resources are supposed unlimited and implementations that oversize the resources available on the device are resolved by temporal partitioning. Then, the temporal partitioned solution is executed by time-sharing the device such that the initial functionality is preserved. This concept promises to be an efficient solution to save silicon area [1]. One of the applications is the switch among functionalities that have mutual exclusiveness on the temporal domain, such as the context-switching between coding/decoding schemes in communication, video or audio systems.

Although, even the latest commercial FPGAs, such as the Xilinx™ Virtex family [3], do not have mechanisms to implement efficiently temporal partitioned functionalities and the time of reconfiguration of the overall FPGA is still quite high, the importance of the "virtual hardware" concept has already been demonstrated with computationally complex applications [4]. Industrial efforts are under way to further improve the capability of the devices to handle multiple-configurations by storing several on-chip configurations and permitting the switch between contexts in few nanoseconds [5].

The virtualization of FPGA resources has been considered by several authors while dealing with circuit netlists that oversize the available resources on the device ([6][7], just to name a few). From the point of view of the design, those approaches work at a much low-level of abstraction, without the possibility to exploit tradeoffs between the number of reconfigurations and the resource sharing of functional units (FUs), for instance. The design automation for FPGA-based systems should include temporal partitioning algorithms able to efficiently exploit the new concept. Tradeoffs among parallelism, communication costs, execution and reconfiguration times, and sharing of some FUs in the same reconfiguration need to be considered during the architectural synthesis phases.

Sharing of FUs among operations is a technique to reuse a single configuration of an FU by more than one operation of the same type. On the other hand, temporal partitioning is a technique tailored to reuse the available resources by different circuits (configurations) with the time-multiplex of the device. The nodes of a given intermediate representation (e.g., a dataflow graph) representing operations have to be scheduled in time steps to be executed in each temporal partition (TP). Temporal partitioning must preserve the dependencies among nodes (that are already temporal dependencies) such that a node **B** dependent on node **A** can-

not be mapped to a partition executed before the partition where node **A** is mapped. In addition, considering sharing FUs during temporal partitioning can conduct to better overall results (lower number of TPs and better performance).

Figure 1a) shows a design flow which integrates temporal partitioning prior to the high-level synthesis tasks [8]. The majority, if not all, of the existent approaches utilizes the presented flow [9][10]. Our efforts address architectural synthesis[1] integrating temporal partitioning and this paper presents a new temporal partitioning algorithm that effectively takes into account sharing of FUs, while maintaining a small computational complexity. Besides, it is sufficiently flexible to target different FPGA devices. Figure 1b) shows the design flow proposed in this paper, where temporal partitioning is integrated in the high-level synthesis tasks and is performed simultaneously.

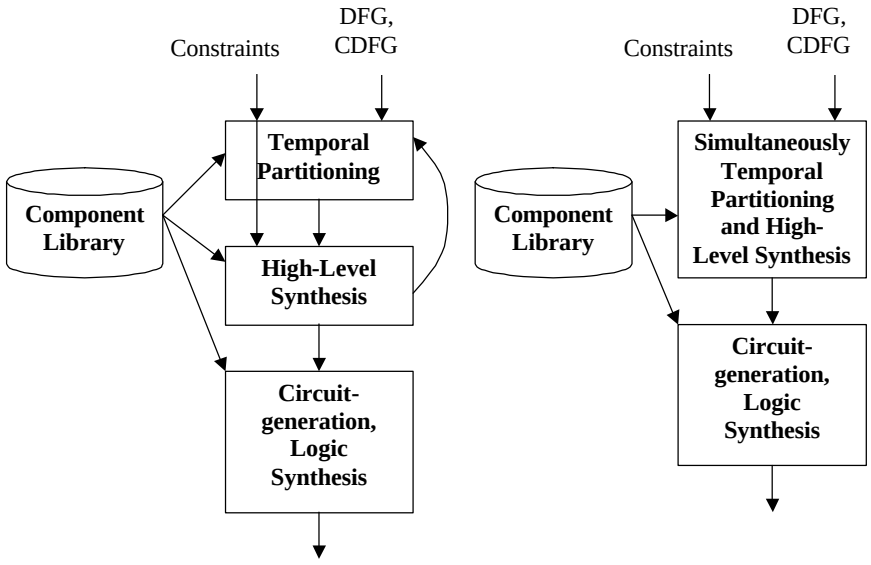

Figure 1. Design flow based on high-level synthesis for reconfigurable computing systems: a) traditional flow; b) proposed flow.

Example 1. Motivational example.

Consider the dataflow graph exhibited in Figure 2 (Ex1). It consists of 4 additions and 2 multiplications. Suppose that each adder uses 1 cell and has a latency of 1 clock cycle, each multiplier uses 2 cells and has a latency of 2 clock cycles and

[1] There is no distinction among the terms: *high-level synthesis, architectural synthesis* and *behavioral synthesis.*

the maximum resources available on the device equals 3 cells. The dataflow graph has a critical path latency of 4 cycles and needs 8 cells given those FUs (last row of Table I). Figure 2 shows an optimal solution (not considering the area of multiplexers, registers and control unit needed to implement sharing of a specific FU) for the example with results shown in the second row of Table I. In Figure 2 each gray region identifies operations that are mapped to the same FU. The optimal solution is achieved with only one adder and one multiplier and fits totally on a single TP. When not considering sharing of adders, the optimum result is shown in the third row of Table I. The algorithm proposed in this paper achieves those optimal results. The fourth row of the table shows the solution obtained when considering a leveling temporal partitioning algorithm that does not consider resource sharing of FUs. From this example, it can be seen that resource sharing can reduce the number of reconfigurations and can also reduce the overall execution latency. There are also cases where the critical path latency of the input dataflow graph (last row) is maintained (second row).

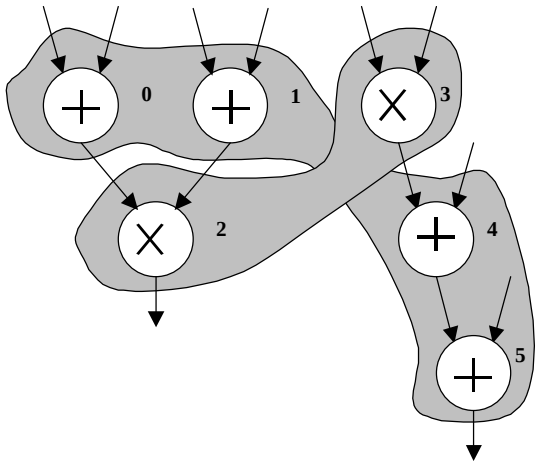

Figure 2. Dataflow graph of the example Ex1.

Table I. Results for Ex1.

Approach	(+, *)	#TPs	Execution latency	Resources used
Optimum (sharing of adders and multipliers)	-	1	4	3
Optimum (sharing of multipliers)	-	3	5	3
ASAP (no sharing) [11]	(4, 2)	4	6	3
Without Temporal Partitioning (no sharing)	(4, 2)	-	4	8

The remainder of this paper is organized as follows. In section 2, previous work is described. Section 3 formulates and explains the problem. The algorithm is deeply explained in section 4, where the pseudo-code and the overall performed steps are fully elucidated through an example. In section 5 experimental results are shown and discussed. Finally, in section 6, conclusions are presented and further work is envisaged.

2 Previous Work

As far as we know, the development of temporal partitioning algorithms was firstly considered in [9][2]. The similarities of both scheduling on high-level synthesis [8] and temporal partitioning allow the use of common scheduling schemes for partitioning. Some authors, such as [9][10], have considered temporal partitioning at behavioral levels having in mind the integration of synthesis.

In [9], a heuristic based on a static list scheduling algorithm, enhanced to consider temporal partitioning and partial reconfiguration, is shown. The approach exploits the dynamic reconfiguration capability of the devices, while doing temporal partitioning.

In [10][12] the temporal partitioning problem is modeled in a specified 0-1 non-linear programming (NLP) model. The problem is transformed to integer linear programming (ILP) and the solution determined by an ILP solver. Due to the long execution times, this approach is not practical for large input examples. Some heuristic methods have been developed to permit its usability on larger input examples [13]. Kaul [14] exploits the loop fission technique while doing temporal partitioning in the presence of loops to minimize the overall latency by utilization of the active TP as long as possible. Sharing of functional units is considered inside tasks and temporal partitioning is performed at the task level. Design space exploitation is performed by inputting to the temporal partitioning algorithm different design solutions for each task. Such solutions are generated by a high-level synthesis tool (constraining the number of FUs of each type). This approach lacks a global view and is time-consuming.

The simplest approaches only consider temporal partitioning without exploiting sharing of FUs. In [11], both a temporal partitioning algorithm based on leveling the operations by an ASAP scheme and other based on clustering a number of nodes are used. The algorithm fills the available resources in the increasing order of the ASAP levels. The selection of nodes in the same level is arbitrary and the algorithm switches to another TP when it encounters the first node that does not fit on the current TP. The approach does not consider neither commu-

nications costs nor resource sharing. In [15] another algorithm is presented that selects the nodes to be mapped in a TP with two different approaches (one for satisfying parallelism and another for decreasing communication costs). In [16], an algorithm based on the extension of the ASAP or ALAP leveling schemes resorting to the mobility of each node to select among the nodes has been considered. [16] also shows an algorithm that searches recursively in the list of ready nodes so that if a node cannot be mapped to the current partition, other nodes can be considered.

[17] considers both communication costs among different TPs that can occur and the overall execution time. The authors presented an extension to static list scheduling, which permits to the algorithm sensitivity to the communication costs while trying to minimize the overall execution time. The results presented, when compared to near-optimal solutions obtained with a simulated annealing algorithm tuned to do temporal partitioning while minimizing an objective function, that integrates the execution time of the TPs and the communication costs, revealed the efficiency of the approach.

[18] presents a method to do temporal partitioning considering pipelining of the reconfiguration and execution stages. The approach divides an FPGA into two portions to overlap the execution of a TP in one portion (previously reconfigured) with the reconfiguration of the other portion.

In [19] constraint logic programming is used to solve temporal partitioning, scheduling, and dynamic module allocation. However, the approach needs a specification of the number of each FU before processing and may suffer of long runtimes.

More related to our approach is the algorithm presented in [20]. A scheme based on the force-directed list scheduling algorithm that considers resource sharing and temporal partitioning is shown. The algorithm tries to minimize the overall execution time, performing a tradeoff between the number of TPs and sharing of FUs. However, the approach adapted a scheduling algorithm not originally tailored to do temporal partitioning and lacks of a global view. Instead, our approach proposes a novel algorithm matched to the combination of temporal partitioning and sharing of FUs that maintains a global view.

3 Problem Definition

Given a dataflow graph (DFG), representing a behavioral description, G = (V, E), topologically ordered, directed and acyclic, with |V| nodes, $\{\nu_1,\nu_2,\ldots,\nu_{|V|}\}$ and |E| edges, where each node ν_i represents an operation and each edge $e_{i,j} \in E$ represents a dependence between

nodes ν_i and ν_j. A dependence can be a simple precedence-dependence or a transport-dependence due to the transport of data between two nodes. The DFG can be obtained from an algorithmic input description. Such pre-processing step is beyond the scope of this article, but the front-end of our Java compiler for reconfigurable computing systems can be employed [16].

Here we assume that there is a component library with a set of FUs and there is one FU for each type of operation in the DFG. Φ represents the set of FUs, from the component library, to be instantiated by the algorithm. R_{MAX} represents the resource capacity available on the device, $R(\pi_i)$ returns the number of resources utilized by the TP π_i and $R(\nu_i)$ returns the number of resources utilized by the FU instance associated with ν_i. $\tilde{A}(\pi_i)$ returns a subset of nodes of V mapped to π_i.

Each partition π_i is a non-empty subset of V, where for each node exists a map to one and only one FU instance in Φ. $\pi(\nu_i)$ identifies the TP where node ν_i is mapped. The set of the TPs is represented by:

$$\wp = \bigcup_{i=1}^{N} p_i \tag{3}$$

where N represents the number of TPs. A graph G, temporal partitioned in N subsets (TPs), is correct if:

- $\bigcap_{i=1}^{N} \Gamma(p_i) = \varnothing$: each node $\nu_i \in V$ is mapped to only one TP (here we do not consider cloning of operations in the DFG);
- $\bigcup_{i=1}^{N} \Gamma(p_i) = V$: all the nodes of V are mapped;
- $\forall \pi_i \in \wp$, $R(\pi_i) \leq R_{MAX}$: each TP fits in the resources available on the device;
- $\forall e_{i,j} \in E$, $\pi(\nu_i) \geq \pi(\nu_j)$: the order of the execution of the TPs does not violate the dependencies among operations of the DFG (necessary condition to obtain the same functionality).

A correct set of TPs guarantees the same overall behavior of the original graph (when executed from 1 to N and considering a correct communication mechanism to transfer data among TPs). However, we are also interested on the minimization of the overall execution latency. The cost that reflects the overall execution latency in a time-multiplexed device can be estimated by the equation (1) or (2), when partial or full reconfiguration of the available re-

sources is considered respectively. CS($\wp$) returns the minimum execution latency (number of control steps or clock cycles, identified as cs) of the partitioned solution, CS(π_i) refers to the minimum execution latency of the TP π_i (it may include the communication costs and represents the execution latency of the critical path of the graph formed by the subset of nodes in π_i and the correspondent edges, considering that nodes sharing FU instances can exist). ∂_i and ∂ represent the number of clock cycles to reconfigure the TP π_i or all the available resources respectively.

$$CS(\wp) = \sum_{i=1}^{N} CS(p_i) + \partial_i \quad (1)$$

$$CS(\wp) = \sum_{i=1}^{N} CS(p_i) + N \times \partial \quad (2)$$

The objective of our algorithm is to furnish a set of datapaths that will be executed in sequence with a minimum number of control steps[2]. Each datapath unit fits on the physically available resources. For the sake of minimizing the number of TPs needed, exploiting sharing of FUs while doing temporal partitioning needs to be considered by the algorithm. Specifically, our algorithm has to output:

- The set of TPs ($\wp$): each TP identifying the nodes of the DFG assigned to it;
- The set of instances for each FU used (Φ);
- Each node of the DFG has to identify a specific FU instance of Φ implementing the operation.

From those outputs, it is straightforward to generate a behavioral HDL-RTL (hardware description language at the register transfer level) description of each TP control unit and a structural HDL-RTL description of each datapath, considering the existence of a HDL description for each FU. The configurations can be generated from those netlists using a traditional FPGA design flow.

4 Simultaneously doing Temporal Partitioning and Sharing of FUs

The algorithm uses an initial number of TPs that can be specified by the user. Another possibility is to use the number of levels of the DFG or the number of TPs utilized by any tempo-

2 - We assume that each control/time step for scheduling is equal to the clock period of the system. Thus, there is no distinction among the use of clock cycle, control step or time step.

ral partitioning algorithm without using sharing of FUs (e.g., ASAP [11]) as the initial number of TPs. The user has to specify the total number of available resources on the device. In addition, for each FU there exists a boolean variable which value indicates if the FU can be shared or not (sharing of some FUs may need more resources than the utilization of several FU instances, due to the overhead of using auxiliary circuits needed for the implementation of the sharing mechanism).

To a clear description, we show the main steps of the algorithm with a connection to Example 1. A brief exposition of the steps performed, when considering sharing of all FUs, is stretched in Figure 3.

The algorithm starts with the following steps:

- Compute the set of nodes child[3] of each node of the DFG;
- Map an FU instance to each operation in the DFG (at the moment neither consider more than one FU for the same operation nor FUs capable to implement more than one operation);
- Estimate the area and execution latency of each node in the DFG according to the FU characterization, existent in the component library, for the target device. This step is beyond the scope of this article and from now on we will assume that there exists, for each FU, an estimation of the number of resources and of the execution latency;
- Perform the ASAP (as soon as possible) and ALAP (as late as possible) start times for each node in the DFG (see Figure 3a)), both unconstrained. When doing the ALAP scheme, the algorithm also calculates the ALAP level of each node;
- Determine the set of nodes in one of the critical paths of the DFG (see Figure 3b));
- Create a number of TPs equal to the input number specified N_{TP}. See the three TPs initially created in Figure 3c);
- Assign each node of the set of nodes in one of the critical paths of the DFG (determined in point 5) to a TP by ascending level. When the number of TPs is larger than the number of nodes in the critical path, the last TPs are left empty; otherwise the last nodes of the set are left unassigned (see the nodes assigned to each TP in Figure 3c));
- Assign the size (number of resources used) of a node in a TP to the current size of that TP (see Figure 3c)).

[3] A node v_i is child of a node v_j if there exists a path from v_j to the end of the DFG that includes v_i.

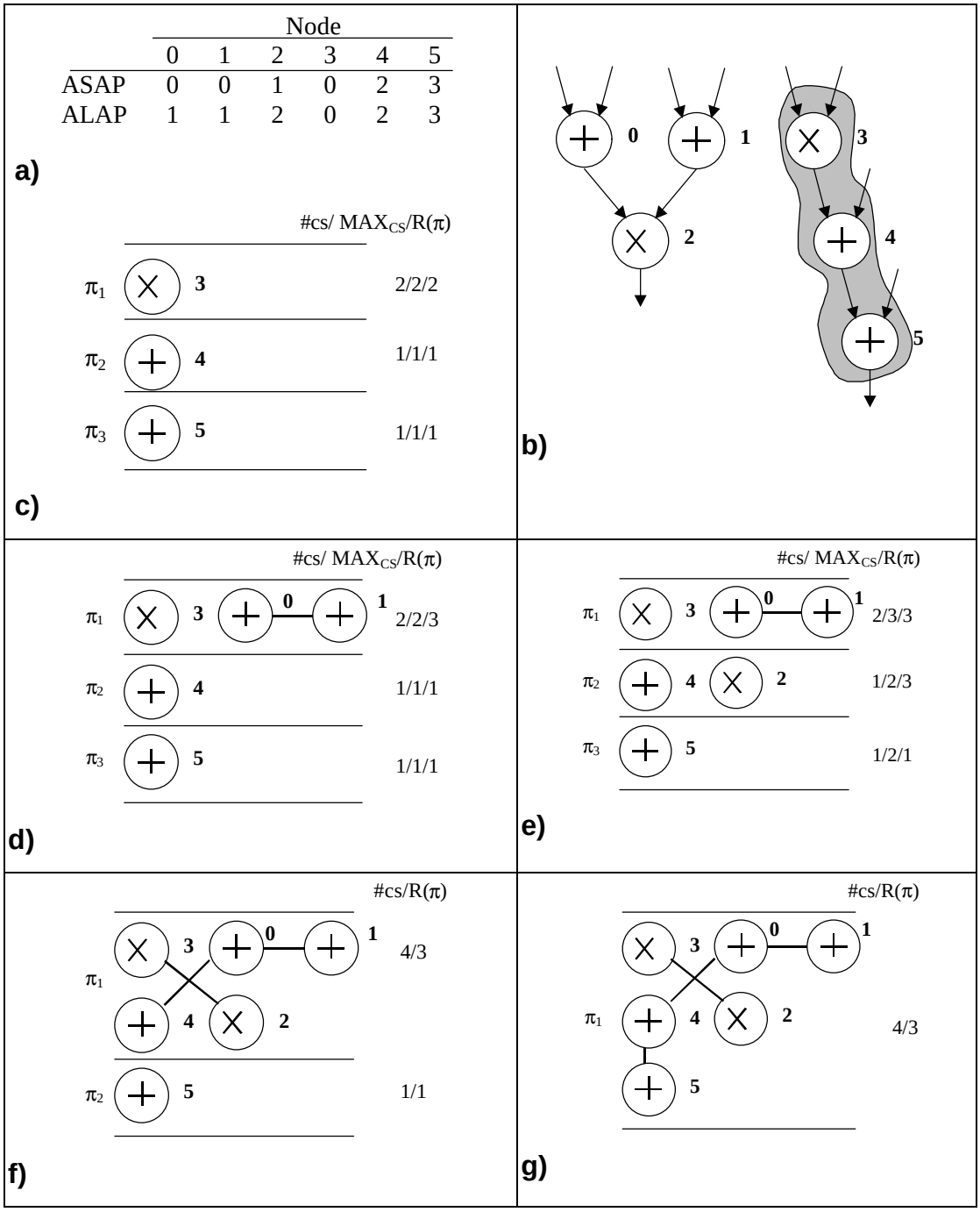

Figure 3. Algorithm execution through an example: a) ASAP and ALAP start times; b) The nodes in the critical path identified by the gray region; c), d), e), f) and g) show iterations of the algorithm.

After the above steps the main kernel of the algorithm is executed (see the pseudo-code in Figure 4, 5 and 6). Some of the most important functions used by the algorithm are listed and briefly explained below:

– v_i.$ALAP_{level}$(): returns the level of v_i considering an ALAP leveling scheme;

– v_i.ALAPStart(): returns the ALAP start time of v_i;

- π_i.addEl(v_i): adds the node v_i to π_i;
- π_i.rmEl(v_i): removes v_i from π_i;
- π_i.sched(v_i): returns the number of control steps of the critical path considering that v_i is mapped to π_i;
- $\wp$.add(π_i): adds a new TP π_i to the current set of TPs (π_i will be the last TP in the set);
- $\wp$.elAt(i): returns the i^{th} TP from the set of TPs ($\wp$);
- *findNodes(i)*: returns a list of nodes ready to be mapped to the i^{th} TP;
- *UpDateAndSortALAP(ListReady, v_i)*: consider the nodes that can be mapped due to the already mapping of node v_i.

Our algorithm will be progressively constructing a global solution. On each iteration, the algorithm traverses the sequence of the existent TPs trying to assign ready nodes to each TP. Each TP has an associated maximum slot time (MAX_{CS}). A node ready to be mapped to a TP is only really considered for mapping if the resultant execution latency of that TP (considering the mapping) does not exceed the correspondent MAX_{CS} (line 15 of Figure 4 and lines 2, 21 and 29 of Figure 5). MAX_{CS} of a given TP π_i is equal to the critical path latency of that TP added by a relax amount: CS(π_i) + relax. On each iteration over the TPs the relax value is incremented by the great common divisor (gcd) among all the execution latencies of the operations in the DFG (line 24 of Figure 4). When a node is mapped (see function *mapNode* in Figure 6), the critical path length of the associated TP is updated (lines 4 and 5 of Figure 6).

The algorithm considers that nodes in contiguous time steps mapped to the same TP and with the same operation should be bound to the same FU instance.

A list of nodes ready for mapping to a current TP is used. The list has the nodes sorted by increasing ALAP start times (the candidate operation having the least ALAP value will have the highest priority) and, for nodes with the same ALAP start time, it uses the ASAP start time as a tiebreak (by ascending or descending order). The list is determined examining for a given node its predecessors (they already must be mapped in TPs before the current TP) and the child set (the nodes child of the node to be mapped must be on TPs after the TP under consideration). The incremental update of the list of the nodes candidate to be mapped to the current TP, when each node is mapped, is an option of the algorithm (lines 6 and 7 in Figure 6). When such option is disabled the algorithm only tries to do update when the list is empty. The algorithm uses a static-based approach in the sense that the ALAP/ASAP values are calculated only once and they are not updated during the execution of the algorithm.

```
1. // begin main kernel
2. BitSet NodesSched = marked with the nodes already mapped to TPs;
3. int NumTP = 0; relax = 0; N = N_TP;
4. int step = gcd(All nodes in DFG);
5. while(notAllNodesSched(NodesSched)) {
6.   LOOP B: while(NumTP < N) {
7.     Vector listReady = findNodes(NumTP);
8.     Vector π_i = ℘.elAt(NumTP);
9.     while(!listReady.isEmpty()) {
10.       Node v_k = listReady.rmFirst();
11.       int R_NEW = R(π_i) + R(v_k);
12.       Boolean fit = (R_NEW <= R_MAX);
13.       // CS(π_i) when v_k is mapped to π_i:
14.       int CS_new = π_i.sched(v_k);
15.       if((CS_new > (CS(π_i)+relax)) && ((π_i is the last TP) && (Γ(π_i) == ∅)
           || (v_k.ALAP_level() < π(v_k)))) {
16.          tryToSched(R_NEW, v_k, fit, CS_new – CS(π_i), π_i, NodesSched, update,
             CS_new, ListReady); Figure 5
17.       } else {
18.          tryToSched(R_NEW, v_k, fit, relax, π_i, NodesSched, update, CS_new,
             ListReady); Figure 5
19.       }
20.     }
21.     NumTP++;
22.   }
23.   NumTP = 0;
24.   relax += step;
25. }
26. // end main kernel
```

Figure 4. Main kernel of the proposed algorithm.

A special directed edge between two nodes is used to identify in the DFG that both nodes share the same FU instance. A path of nodes connected by such edges identifies the sequence of utilization of the FU instance (from the source to the sink). Those edges are added by the algorithm to the initial DFG and provide an efficient way to both represent sharing of FU instances, to determine the execution latency and to generate the datapath and control unit descriptions. When a node v_n is bound to an FU instance that is already shared by two or more operations, the algorithm adds a special edge from the last node of the path of shared FUs to v_n. However, when there are one or more nodes in that path that are child of v_n in the DFG, v_n is inserted immediately before the first child found traversing the path from left to right (see Figure 7).

The algorithm considers the interchange[4] of a node previously assigned to a TP with a node ready to be mapped in that TP. This occurs if the TP has only one node, the node to be

[4] This involves the move of a node from a TP to the list and the map of the first node ready to that TP.

mapped has lower ALAP start time and the change is feasible in terms of available resources and MAX_{CS} (lines from 3 to 13 of Figure 5).

```
1. tryToSched(int R_NEW, Node vi, Boolean fit, int relax, TP π_k, BitSet
   NodesSched, Boolean update, int CS_new, ListReady) {
2.   if((CS_new ≤ (relax+CS(π_k)) || (CS(π_k) == 0)) {
3.     if only one node v_j in π_k {
4.        if(v_j.ALAPStart() > v_i.ALAPStart()) {
5.          if((R_NEW – R(v_j)) <= R_MAX) {
6.             π_k.rmEl(v_j);
7.             R(π_k) = R(v_i);
8.             π_k.addEl(v_i);
9.             NodesSched.clear(v_j);
10.            NodesSched.set(v_i);
11.            continue LOOP B;
12.         }
13.       }
14.    }
15.    boolean canShare = try sharing with a node of the same type with a
       path of shared FUs with the smallest length (number of nodes;
16.    if(canShare) {
17.      if(fit && share produces increase) {
18.         canShare = false;
19.         rmShare(v_i);
20.      } else {
21.         int CS_new1 = π_k.sched(v_i);
22.         if(CS_new1 ≤ (relax + CS(π_k))) {
23.            mapNode(π_k, v_i, NodesSched, update, CS_new1, ListReady); Figure
               6
24.         } else {
25.            rmShare(v_i);
26.            canShare = false;
27.         }
28.      }
29.    }
30.    if(!canShare && fit && (CS_new ≤ (relax+ CS(π_k))) || (CS(π_k) == 0)) {
31.      mapNode(π_k, v_i, NodesSched, update, CS_new, ListReady); Figure 6
32.    }
33.    if(v_i not mapped and no FU with operation type of v_i in thisTP and v_i
       does not fit and thisTP is the last TP) {
34.       create a new TP π_n; N++;
35.       ℘.add(π_n);
36.       mapNode(π_n, v_i, NodesSched, update, CS(π_n), ListReady); Figure 6
37.       break LOOP B;
38.    }
39.  }
40.    if((CS(π_k) + relax) ≤ MaxLatOp) {
41.    if(!Share && !wasSched) break LOOP B;
42.  }
43. } // end tryToSched
```

Figure 5. Function tryToSched.

```
1. mapNode(TP πk, Node vi, BitSet NodesSched, Boolean update, int CSnew, Vec-
   tor ListReady) {
2.   πk.addEl(vi);
3.   NodesSched.set(vi);
4.   if(CSnew > CS(πk))
5.     CS(πk) = CSnew;
6.   if(update)
7.     upDateAndSortALAP(ListReady, vi);
8. } // end mapNode
```

Figure 6. Function mapNode.

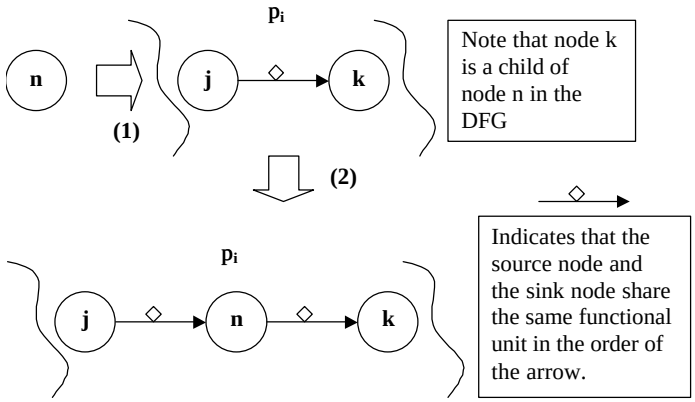

Figure 7. Special mapping of a node in a TP.

After the execution of the main kernel the algorithm considers a merge operation. Such operation tries to group adjacent TPs in a single TP considering resource sharing among FU instances of the same type in two considered TPs (see Figure 3e), f) and g)). This step of the algorithm is done incrementally and it stops when no merging is possible. From the two first TPs of Figure 3e) it can be seen that sharing the FUs, the two TPs can be merged. Figure 3f) illustrates the result of that merge. From the later result we can figured out that π_1 and π_2 are also able to be merged (with node 5 sharing the FU instance already shared by nodes 0, 1 and 4). The resultant merge is shown in Figure 3g) and the final solution requires only a single TP.

When a node does not fit (in the sense that the MAX_{CS} of the considered TP is violated) in the last TP and if that TP is empty then the algorithm prefers to relax the TP to accommodate the node (see lines 15 to 17 in Figure 4).

When finding if a node to be mapped can share an FU instance already existent in the considered TP, the algorithm binds the node/operation to the FU instance with lower length path of nodes sharing it (line 14 in Figure 5).

The steps of the algorithm shown in lines 32 to 37 of Figure 5 are related to the case that when a node does not fit (with or without resource sharing or with relaxation of MAX_{CS}) in the last TP a new TP is added to the existent set of TPs and the node is mapped to it.

The algorithm optionally uses a jump to the first TP (lines 38 to 40 of Figure 5). This forces the algorithm to consider the binding of nodes, which cannot share an FU in the current TP, to earlier TPs.

4.1 Time Complexity of the Algorithm

The worst time complexities for each step of the algorithm are:

- $O(|V|+|E|)$, for ASAP and ALAP schemes;
- $O(|V|+|E|)$, for determination of the child nodes presented in all the paths from each node of the DFG to the end, when the DFG is topologically ordered;
- $O(|V|^2)$, for sorting a list of $|V|$ nodes;
- $O(|V|)$, for searching a node in each TP;
- $O(N_{PASS}.N.|V|)$, for the core of the algorithm, where N_{PASS} is the number of passes of the algorithm through the N TPs. N_{PASS} has a majoring value of $\lceil |V| \times MaxLatOp/\text{step} \rceil$, where *MaxLatOp* represents the largest latency of the set of FUs that could be assigned to nodes of the DFG;
- $O(N.|V|)$, for the merge operation.

Thus, the worst time complexity of the algorithm is $O(k.N.|V|^2)$, with k equal to *MaxLatOp*/step, or just $O(N.|V|^2)$.

5 Experiments and Results

All the algorithms considered in this section have been implemented with the Java™ language. All the executions of the algorithms were conducted on a portable PC (Pentium-II @366MHz, 196Mb RAM) with the JIT compiler integrated in the JDK1.2 running on Windows98.

The algorithm proposed in this paper has been tested with a number of representative examples (with variable complexity) and, whenever possible, the results obtained are compared with other approaches.

5.1 Examples

The SEWHA is the auto regression filter presented in [21], HAL is the loop body of the differential equation example [22], EWF is a digital fifth order elliptic wave filter [23], FIR is a 12-tap finite impulse filter, Mat4×4 corresponds to a fully parallel multiplication of two matrixes with 4×4 integer elements each one, and FDCT is the fast discrete cosine transform used in [24]. Mat4×4 is used as an example of high operation level parallelism degree.

We consider for all the experiments an execution latency of 1 clock cycle for each adder or ALU and 2 clock cycles for each multiplier (only nonpipelined multipliers were used). For the number of resources needed for each FU we consider 1 cell for each adder or ALU and 4 cells for each multiplier. Table II shows the main characteristics of the considered examples (number of operations of each type, total number of resources and the critical path length).

Table II. Characteristics of the examples.

Example	Number of operations (×, +/-)	Total number of operations	Total number of resources (cells)	Critical path length of the DFG (clock cycles)
Ex1	(2, 4)	6	12	4
EWF	(8, 26)	34	58	17
FIR	(12, 11)	33	59	6
HAL	(4, 6)	10	22	7
SEHWA	(16, 12)	28	76	11
Mat4×4	(64, 48)	112	304	4
FDCT	(16, 26)	42	90	8

5.2 Sharing versus not sharing

Table III shows results for the considered examples. The ASAP results refer to the leveling technique proposed in [11]. The SA results were obtained with a simulated annealing approach to do temporal partitioning without resource sharing proposed in [17]. Here, the algorithm is tuned to optimizing the overall execution time without entering into account the communication costs (the algorithm can also exploit the tradeoff between execution time and communication costs). Values shown represent the best results collected from several runs of the algorithm (minutes and hours of CPU time).

Our(1), (2) and (3) identify results obtained by applying the proposed algorithm. Our(1) identifies executions of the algorithm disabling the capability to share FUs among operations. Our(2) considers resource sharing for both adder and multiplier units, and Our(3) only considers resource sharing for multiplier units. #cs identifies the execution latency (number of

clock cycles) and #p the number of TPs. The results for our algorithm refer to solutions with the minimum number of control steps obtained (not considering the time to reconfigure). Such solutions do not necessarily have the minimum number of TPs. Each solution related to our algorithm was obtained in less than 1s of CPU time.

Only Mat4×4 needed to start with the number of TPs obtained by the ASAP approach to achieve the best solution. For all the other examples, the best solution was obtained starting with an initial number of TPs equal to the number of levels of the DFG. The results for Mat4×4 in Table III were collected disabling the update of the list of nodes ready for each node mapped (the list is updated only when it is empty). It is strongly recommended to disable the update option for examples with high-level degree of parallelism and a small critical path length.

Table III. Results obtained for the examples.

Example	R_{MAX}	Approach									
		ASAP w/o sharing		**SA** w/o sharing		**Our(1)** w/o sharing		**Our(2)** w/ sharing of +'s, ALU's and ×'s		**Our(3)** w/ sharing of ×'s	
		#p	#cs	#p	#cs	#p	#cs	#p	#cs	#p	#cs
Ex1	6	2	5	2	**4**	2	**4**	1	**4**	2	**4**
SEHWA (AR)	6	18	36	18	35	17	38	1	**34**	6	**34**
	10	9	24	9	19	9	20	1	**18**	6	**18**
	15	6	18	6	**15**	7	**15**	1	**15**	5	**15**
HAL	6	5	11	5	**9**	4	**9**	1	**9**	3	10
	10	3	10	3	**7**	3	**7**	2	**7**	3	**7**
EWF	6	12	26	12	**22**	12	25	1	23	9	25
	10	6	22	6	21	8	**19**	5	**18**	7	**18**
	15	4	19	4	18	4	**18**	1	**17**	4	18
FIR	6	14	28	14	**27**	13	28	1	**20**	4	27
	10	7	16	7	**15**	7	**15**	1	**15**	4	**15**
	15	5	12	5	**11**	5	12	1	**11**	3	**11**
Mat4×4	6	72	136	72	**130**	64	**130**	1	**130**	21	**130**
	10	37	69	37	**66**	33	**66**	1	**66**	17	**66**
	15	25	47	25	**46**	22	**46**	1	**46**	10	**46**
	20	16	29	16	**29**	16	**29**	2	**29**	4	**29**
FDCT	6	20	36	20	**34**	18	36	2	**34**	6	36
	10	11	21	11	**19**	10	20	2	**19**	5	20
	15	7	17	7	**14**	7	15	3	**14**	6	**14***
	20	5	14	5	**12**	5	**12**	2	**12**	5	**12**

The values in bold in the 4th, 6th, 8th, 10th and 12th columns of Table III show the minimum execution latency for the datapaths obtained by the considered approaches (not considering configuration times). The 8th column presents results when executing our algorithm disabling

the sharing of all FUs. Twelve cases, represented in bold, are not worse to the results close-to-optimum produced with SA. Surprisingly, one case obtained by Our(1) was not achieved by SA. The values in bold in the 12th column represent that, even without considering sharing of adders, our algorithm returns solutions with execution latencies equal to the execution latencies obtained sharing all the resources (10th column), despite the fact that those solutions need more TPs. When compared to ASAP our algorithm produces only 1 worst case.

When considering resource sharing for all FUs, a minimum number of TPs (only 7 cases of Table III needed more than one TP to produce a minimum execution time) seems to ensure solutions with lower execution latencies than the obtained by doing temporal partitioning with ASAP or SA for the majority of the examples (5 cases are better than the close-to-optimum SA results). Note that when all the FUs can be shared and the resource overhead to implement sharing is not taken into account, an empirical observation tell us that, most of times, the solutions with lower execution latency are those with only one TP. This is expected by the fact that a new TP produces an equal or worse effect than sharing FU instances on the overall execution latencies because all the nodes in that TP can only start executing after the end of the execution of the TP immediately before.

When sharing of adders is not considered the algorithm is capable to find 14 solutions without inferior execution latency.

5.3 Exploiting the number of TPs

An exploitation of the overall execution latency versus the number of TPs is, for the Mat4×4 example, shown in Figure 8. Those results were produced by calling the algorithm several times, each time starting with a different initial number of TPs from a range of 1 to 15. The exploitation has been done in approximately 5.4s of CPU time. All the solutions use only a single TP and the best result (execution latency equal to 66 clock cycles) has been achieved when the algorithm started with 8 TPs. The results without considering sharing of adders are shown in Figure 9. The algorithm exploited a range of TPs from 1 to 26 and the minimum execution latency achieved was 66 clock cycles (solution with 21 TPs). Based on those results we can select a solution that minimizes the global execution latency taking into account the reconfiguration times (see equation (2)).

From the results presented so far we may conclude that sharing FUs can reduce the number of TPs without increasing the overall execution time. Moreover, a minimum number of TPs can be a priority, when an FPGA with significant reconfiguration times is used. Due to its

low computational complexity, the algorithm can be used to exploit the design space based on the tradeoff between the number of TPs and the overall execution latency.

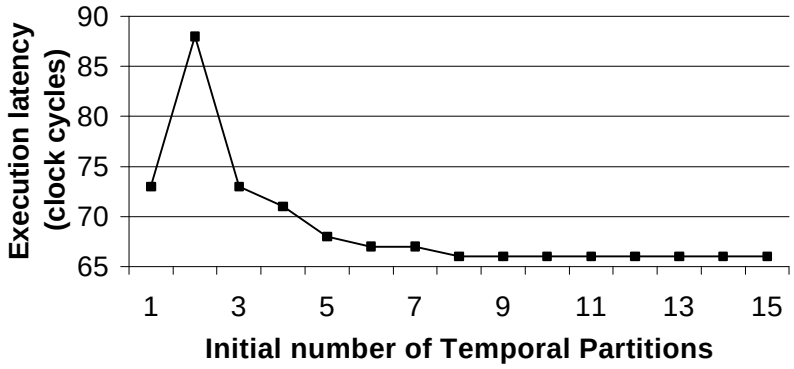

Figure 8. Execution latency versus the initial number of TPs for Mat4´4 obtained by the proposed algorithm, when R_{MAX}=10 (sharing of adders and multipliers).

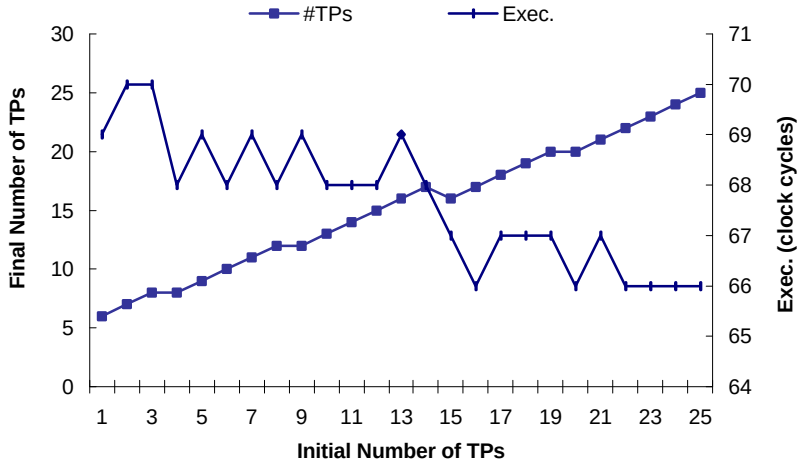

Figure 9. Execution latency and the final number of TPs versus the initial number of TPs obtained by the algorithm for Mat4´4, when R_{MAX}=10 (no sharing of adders).

5.4 Comparison with schedulers

At this point a question may occur: is the algorithm competitive when a single TP is envisaged? Table IV shows results for EWF and SEWHA, considering various sizes for the available resources (R_{MAX}). The schedule lengths obtained by the proposed algorithm considering only one TP are shown (see the 5th column). The number of resources used for each type of

FU for each solution is also shown (6th column). "Fixed" refers to results collected from the state-of-the-art schedulers [25][26][27] and represent optimal (identified with ♣) or near-optimal scheduling results (without taken into account to temporal partitioning) for the specified constraint on the number of FUs for each type of operation (see the 2nd column). The results show that our algorithm is efficient, even when we are interested on a final solution with a single TP.

Table IV. Comparison of scheduling results obtained for EWF and SEHWA.

Example	Approach					#cs difference
	Fixed [25][26][27]		Our			
	constraints (×, +)	#cs	constraints R_{MAX} (#p=1)	#cs	(×, +)	
EWF	(1, 2)	21♣	6	22	(1, 2)	+1
	(1, 3)	18	7	22	(1, 3)	+4
	(2, 1)	21♣	9	22	(1, 5)	+1
	(2, 2)	18♣	10	20	(2, 2)	+2
	(2, 3)	18	11	**18**	(2, 3)	0
	(3, 2)	18	14	**18**	(2, 6)	0
	(3, 3)	17♣	15	**17**	(3, 3)	0
SEHWA (AR)	(1, 1)	34♣	5	**34**	(1, 1)	0
	(2, 1)	18♣	9	20	(2, 1)	+2
	(2, 2)	18	10	**18**	(2, 2)	0
	(3, 1)	16♣	13	17	(3, 1)	+1
	(3, 2)	15♣	14	**15**	(3, 2)	0
	(3, 3)	15♣	15	**15**	(3, 3)	0
	(4, 1)	16♣	17	**16***	(4, 1)	0
	(4, 2)	11♣	18	**11**	(4, 2)	0

The result labeled with a "*" is achieved without an incremental update of the list of the nodes ready to be mapped. This result shows that the algorithm did not skip from a local minimum, since at least the result related to R_{MAX}=15 should be achieved.

Some results consider the increasing order of the ASAP values as the second key (there is no evidence to suggest when it is better to use the decreasing or the increasing ASAP values as the second key).

The number of each FU instance allocated by our algorithm for each R_{MAX} constraint only was different in two cases to the constraints used (with total number of resources equal to R_{MAX}) to produce the near-optimal scheduling results (see Table IV). Therefore, it seems that our algorithm can also be used to a fast identification of the number of FU instances needed, considering a specific number of maximum resources available on the device.

6 Conclusions and Future Work

In this paper we have presented a new and useful algorithm combining temporal partitioning, sharing of functional units, scheduling, allocation and binding. Unlike other approaches, this algorithm merges those tasks in a combined and global method. The obtained results, from a number of benchmarks, strongly confirm the efficiency and effectiveness of the idea.

The low computation time achieved, when dealing with the presented examples, shows that the algorithm is fast and efficient and thus can be used on large examples.

Work in progress focuses practical extensions to the algorithm. Such extensions are taking into account the hardware resources - for multiplexers, registers and for the control unit - needed for sharing a functional unit.

Extensions to deal with functional units with pipeline stages and with more than one implementation for a given operation will be considered in a near future. Another important issue is the overlapping of reconfiguration and execution that should be considered by future enhancements. Finally, aspects related to conditional paths and loops will also need to be focused on future work.

7 Acknowledgments

The author would like to acknowledge the support from the PRAXIS XXI Program under the scope of the AXEL Project (PRAXIS/P/EEI/12154/1998).

8 References

[1] A. DeHon, *Reconfigurable Architectures for General Purpose Computing*, PhD Thesis, AI Technical Report 1586, MIT, 545 Technology Sq., Cambridge, MA02139, Sept. 1996, http://www.ai.mit.edu/people/andre/phd.html.

[2] X.-P.Long, H. Amano, "WASMII: a Data Driven Computer on a Virtual Hardware," in *Proc. of the 1st IEEE Workshop on Field Programmable Custom Computing Machines (FCCM'93)*, Napa Valley, CA, USA, April 5-7, 1993, pp. 33-42.

[3] Xilinx Inc., *Virtex Field Programmable Gate Arrays*, version 1999.

[4] R. D. Hudson, D. I. Lehn, and P. M. Athanas, "A Run-Time Reconfigurable Engine for Image Interpolation," in *Proc. of the 6th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'98)*, Napa Valley, CA, USA, April 15-17, 1998,

Kenneth L. Pocek and Jeffrey Arnold, editors, IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 88-95.

[5] T. Fujii, et al, "A Dynamically Reconfigurable Logic Engine with a Multi-Context/Multi-Mode Unified-Cell Architecture," in *Proc. of the IEEE Int'l Solid State Circuits Conference (ISSCC'99)*, SA, CA, Feb. 15-17, 1999, pp. 364-365. Available online at: http://www.isscc.org/digests/1999/DATA/21_3.pdf.

[6] S. Trimberger, "Scheduling Designs into a Time-Multiplexed FPGA," in *Proc. of the 6th ACM Int'l Symposium on Field Programmable Gate Arrays (FPGA'98)*, Monterey, CA, USA, February 22-24, 1998, pp. 153-160.

[7] H. Liu and D. F. Wong, "Circuit partitioning for dynamically reconfigurable FPGAs," in *Proc. 7th ACM Int'l Symposium on Field Programmable Gate Arrays (FPGA'99)*, Monterey, CA, USA, Feb. 21-23, 1999, pp. 187-194.

[8] D. Gajski, et al., *High-Level Synthesis, Introduction to Chip and System Design*, Kluwer Academic Publishers, 1992.

[9] M. Vasilko, D. Ait-Boudaoud, "Architectural Synthesis Techniques for Dynamically Reconfigurable Logic," in *Proc. of the 6th Int. Workshop on Field-Programmable Logic and Applications (FPL'96)*, Darmstadt, Germany, Sept. 23-25, 1996, LNCS, vol. 1142, Springer-Verlag, pp. 290-296.

[10] I. Ouaiss, et al., "An Integrated Partitioning and Synthesis System for Dynamically Reconfigurable Multi-FPGA Architectures," in *Proc. of the 5th Reconfigurable Architectures Workshop (RAW'98)*, Orlando, Florida, USA, March 30, 1998, pp. 31-36.

[11] Karthikeya M. GajjalaPurna, and Dinesh Bhatia, "Temporal Partitioning and Scheduling Data Flow Graphs for Reconfigurable Computers," in *IEEE Transactions on Computers*, vol. 48, no. 6, June 1999, pp. 579-591.

[12] M. Kaul, R. Vemuri, "Optimal Temporal Partitioning and Synthesis for Reconfigurable Architectures," in *Proc. of the Design, Automation & Test in Europe (DATE'98)*, Paris, France, Feb. 23-26, 1998, pp. 389-396.

[13] M. Kaul, R. Vemuri, "Temporal Partitioning combined with Design Space Exploration for Latency Minimization of Run-Time Reconfigured Designs," in *Proc. of Design, Automation & Test in Europe (DATE'99)*, Paris, France, Feb. 23-26, 1999, pp. 202-209.

[14] Meenakshi Kaul, Ranga Vemuri, Sriram Govindarajan, Iyad E. Ouaiss, “An Automated Temporal Partitioning and Loop Fission approach for FPGA based reconfigurable synthesis of DSP applications,” in *Proc. of the IEEE/ACM Design Automation Conference (DAC'99)*, New Orleans, LA, USA, June 21-25, 1999, pp. 616-622.

[15] Atsushi Takayama, Yuichiro Shibata, Keisuke Iwai, and Hideharu Amano, “Dataflow Partitioning and Scheduling Algorithms for WASMII, a Virtual Hardware,” in *Proc. of the 10th Int. Conference on Field-Programmable Logic and Applications (FPL’00)*, Villach, Austria, August 27-30, 2000. Reiner W. Hartenstein and Herbert Grünbacher (eds.), LNCS, vol. 1896, Springer-Verlag, Berlin, pp. 685-694.

[16] João M. P. Cardoso, and Horácio C. Neto, “Macro-Based Hardware Compilation of Java™ Bytecodes into a Dynamic Reconfigurable Computing System,” in *Proc. of the IEEE 7th Symposium on Field-Programmable Custom Computing Machines (FCCM'99)*, Napa Valley, CA, USA, April 21-23, 1999, Kenneth L. Pocek and Jeffrey Arnolds (eds.), IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 2-11.

[17] João M. P. Cardoso, and Horácio C. Neto, “An Enhanced Static-List Scheduling Algorithm for Temporal Partitioning onto RPUs,” in *Proc. of the IFIP X International Conference on Very Large Scale Integration (VLSI'99)*, Lisbon, December 1-3, 1999. Luis M. Silveira, Srinivas Devadas and Ricardo Reis (eds.), *VLSI: Systems on a Chip*, Kluwer Academic Publishers, pp. 485-496.

[18] Satish Ganesan, and Ranga Vemuri, “An Integrated Temporal Partitioning and Partial Reconfiguration Technique for Design Latency Improvement,” in *Proc. of Design, Automation & Test in Europe (DATE'00)*, Paris, France, March 27-30, 2000, pp. 320-325.

[19] Xue-jie Zhang, Kam-wing Ng, and Wayne Luk, “A Combined Approach to High-Level Synthesis for Dynamically Reconfigurable Systems,” in *Proc. of the 10th Int. Conference on Field-Programmable Logic and Applications (FPL’00)*, Villach, Austria, August 27-30, 2000. Reiner W. Hartenstein and Herbert Grünbacher (eds.), LNCS, vol. 1896, Springer-Verlag, Berlin, pp. 361-370.

[20] Awartika Pandey and Ranga Vemuri, “Combined Temporal Partitioning and Scheduling for Reconfigurable Architectures,” in *Proc. SPIE Photonics East Conference, SPIE 3844*, Boston, Massachusetts, USA, Sept. 20-21, 1999. John Schewel, et al. (eds.), *Reconfigurable Technology: FPGAs for Computing and Applications*, pp. 93-103.

[21] R. Jain, A. Parker, N. Park, “Module Selection for Pipelined Synthesis,” in *Proc. of the 25th Design Automation Conference*, Anaheim, CA, USA, June 12-15, 1988, pp. 542-547.

[22] P. G. Paulin, J. P. Knight, and E. F. Girczyc, “HAL: A Multi-Paradigm Approach to Automatic Data Path Synthesis,” in *Proc. of the 23rd Design Automation Conference*, Las Vegas, NV, USA, June 29-July 2, 1986, pp. 263-270.

[23] P. Dewilde, E. Deprettere, and R. Nouta, “Parallel and Pipelined VLSI Implementation of Signal Processing Algorithms,“ in *VLSI and Modern Signal Processing*, S.Y. Kung, H.J. Whitehouse, T.Kailath (eds.), Prentice-Hall 1985, pp. 258-264.

[24] D. J. Mallon, and P. B. Deneyer, “A new approach to pipeline optimization,“ in *Proceedings of the European Design Automation Conference (EDAC’90),* March 1990, pp. 83-88.

[25] M. Narasimhan, and J. Ramanujam, “A Fast Approach to Computing Exact Solutions to the Resource-Constrained Scheduling Problem, “ to appear in *ACM Transactions on Design Automation of Electronic Systems (TODAES),* Vol. 7, No. 1, January 2002. URL: http://www.acm.org/todaes/V7N1/TOC.html

[26] P.-Y. Hsiao, G.-M. Wu, and J.-Y. Su, “MPT-based branch-and-bound strategy for scheduling problem in high-level synthesis,” in *IEE Proc. Computers and Digital Techniques*, Vol. 145, No. 6, November 1998, pp. 425-432.

[27] M. K. Dhodhi, F. H. Hielscher, R. H. Storer, and F. Bhasker, “Datapath Synthesis Using a Problem-Space Genetic Algorithm,” in *IEEE Transactions on CAD of Integrated Circuits and Systems*, Vol. 14, No. 8, August 1995, pp. 934-944.