

Virtualization of FPGA via Segmentation

William Fornaciari, Vincenzo Piuri, Luigi Ripamonti

Department of Electronics and Information, Politecnico di Milano

Piazza L. da Vinci 32, 20133 Milano, Italy

Phone +39-02-2399-3623, Fax +39-02-2399-3411, Email {fornacia, piuri}@elet.polimi.it

Abstract

Recent advances in RAM-based FPGAs technologies contribute to increase their use as an efficient and flexible computational support for real-time applications, whenever ASIC solution are too expensive or the time-to-market is too critical. Nevertheless, despite the steady improvement of the configuration time, the physical dimensions of FPGAs (i.e., the number of I/O resources as well as the number of CLBs) limit the complexity of circuits that can be implemented, making mandatory the choice of multi-FPGA systems for large systems.

An alternative approach to cope with cost of the FPGA systems when very high performance is not the primary goal, is circuit partitioning with part multiplexing on a single FPGA. Dynamic loading, pagination, and segmentation (already used in operating systems) are techniques that allow for creating the *Virtual FPGAs* in multitasking computing environments. This paper deals specifically with segmentation and the related problems: possible strategies and designer's guidelines to perform segmentation effectively are the paper focus.

1. Introduction

FPGAs are programmable devices composed by a number of different configurable resources [1-4]: the *logic cells* (possibly containing memory elements) which realize the combinatorial or the sequential logic of the circuit, the *interconnection resources* which allow the information exchange among different cells, and the *input and output pins* which dispatch the input data and read out the computation results.

RAM-based FPGAs can be adopted to realize stand-alone dedicated combinatorial or sequential circuits or simply as an ideal medium for prototyping; moreover, they may be used as co-processing units in general-purpose microprocessor-based systems. On the market nowadays are also available FPGA-based boards for standard personal computers; this availability allows the designer to decide if the algorithms can be executed either in software or in hardware, namely on the FPGA, yielding a great improvement on global performances. However, some physical limitations exist in the use of FPGAs, especially due to the limited number of configurable resources that reduces the set of applications that can be implemented in a multitasking system, making mandatory the use of a software solution. Alternatively, a conventional solution consists of partitioning the application circuit and in mapping each part on a different FPGA, leading to the so-called multi-FPGA systems [5,6]. On the other hand, some algorithms may be much smaller than the target FPGA yielding a sensible waste of resources.

In order to avoid such problems, we can adopt some techniques which are borrowed from modern operating systems [7]:

- *dynamic loading*. The FPGA configuration is modified as dynamically required by the concurrent tasks [8,9]. When a single application task is using the FPGA, this technique is also well known as run-time reconfiguration [10-15].
- *static/dynamic partitioning*. The FPGA resources are grouped into disjoint sets, each assigned to a different task, yielding a much more efficient utilization of the device and a reduced number of loading and storing of different configurations [16].
- *segmentation*. Circuits, that require resources exceeding the available ones, are physically decomposed by the designers into smaller partitions, sequentially loaded onto the FPGA, in order to recreate the computation as if all of them were loaded into the device at the same time.

This paper deals with segmentation as an effective solution to load parts of the application circuits in a multitasking environment, in which several concurrent tasks need to use the shared FPGA device. Our goal is to support concurrent execution of several application circuits and allow implementation of circuits that exceed the size of the physical FPGA device.

In the available literature, the dynamic configuration has been widely studied to provide flexibility to the implemented functions [9] or to overcome the size limits of the physical FPGA [10-12] when only *one* application task controls the sequence of FPGA configuration loading. The case of multitasking has been considered by assuming that each individual application circuit does not exceed the resources available in the FPGA [13], being each of them mapped alternatively in the FPGA device. The case of multitasking with different application circuits that are loaded in parallel in the shared FPGA is considered in [14], but they are loaded entirely; in [15] only partial reconfiguration is allowed. Our approach is therefore more general and open new possibilities for advanced configurable computing machines, even on commercially-available general-purpose machines with FPGA boards.

The proposed basic technique for segmented FPGAs is introduced in Section 2. The design and operation issues are discussed to provide effective design guidelines in Section 3.

2. Segmentation of FPGA

In the literature on operating systems (e.g., [7]), segmentation is defined as a technique for splitting a program into parts so that it can be executed on computers with less physical memory. In fact, because of locality of memory accesses, there is no actual need for all parts at the same time. This allows the operating system to load the different segments as needed and to unload the ones not useful for the current computation, thus freeing memory for others. Segments generally contain strictly related procedures or data: The programmer is responsible for identifying them.

Segmentation is more difficult to define in digital circuits. In fact, even if it is still possible to identify parts of the circuit which are strictly related each others (e.g., the different VHDL components linked by a hierarchical relationship), the designer may find some difficulties in clustering the components into different segments in an unambiguous way, especially when cycles among components are present.

Partitioning may be performed in many different ways. For example, it may be accomplished onto a gate-level representation of the circuit. In some cases, this leads to good results even if partitioning must be fully automated due to the complexity of the circuit representation and the low abstraction level of the description itself, thus lacking a direct control of the designer onto the synthesis process.

Otherwise, we can work on the VHDL representation of the circuit to be partitioned [17]. In this case segmentation is performed at a high abstraction level. Besides, the VHDL language allows to define hierarchical relationships among components. For the designer it becomes therefore easier to identify ‘procedures’ and ‘sub-procedures’ in a similar manner to that used for software programs.

Nevertheless, there are some drawbacks with the VHDL approach. First, the need of estimating the size of each VHDL component, e.g., by adopting the FPGA mapping tool or by associating a size to each VHDL constructor and parse each files composing the circuit. Second, designers may find some difficulties in creating segments while respecting the FPGA size constraint and not wasting configurable resources excessively.

Since the FPGA is time-shared among different segments and, possibly, among different tasks, saving and, then, restoring the state of each segment (i.e., the values stored in each memory element of the segment) is required. If the target FPGA does not provide direct access to these data from outside, an auxiliary circuit for each segment is required to support the state management.

Management of input and output data for each segment is another problem we need to face. Generally only one segment per process is loaded into the FPGA. As a consequence, there are no direct, physical connections among different segments of the same circuit. Therefore, we need a mechanism for saving the segment outputs and, then, delivering them to the correct subsequent segments. Also this problem can be solved either by adopting an auxiliary circuit or by saving all data in the central memory. In the first case, the auxiliary circuit is global to the whole application circuit and must be permanently loaded into the FPGA.

2.1 State information management

To save and restore the segment state information two approaches can be envisioned, according to the structural characteristics of the FPGA: introducing an auxiliary circuit in the application circuits to extract

and hold the state information, or direct reading and writing the memory elements holding the state information within the segments.

The auxiliary circuit becomes necessary when the FPGA does not provide a build-in mechanism to access directly to the values stored in the possible memory element of each CLB, by using the dedicated input/output pins which are available in every FPGA device to save and restore data. This auxiliary circuit links each state bit to an I/O pin, with suited circuitry to control reading and writing from outside the FPGA. The size of this circuit is related to the approach adopted to implement its operation and to the total number of memory elements to be managed. Such a circuit need to be present in the FPGA at any time to correctly transfer the state information at the proper time. This solution reduces the FPGA resources available for the segments of the application circuits and, in turn, the number of segments produced by partitioning increases with the number of memory elements. The unavailable resources are both CLBs and I/O pins.

The size of the auxiliary circuit needs to be estimated quite accurately during partitioning to avoid to violate the constraint imposed by the FPGA device on the sub-circuit size. A first approach works at high abstraction level on the VHDL files, before partitioning is actually started. We can in fact modify each behavioral component of the VHDL circuit description by physically including the state management circuit as a local component. Information needed for size estimation are thus implicitly available during partitioning since the auxiliary circuit will be analyzed as any other circuit. It is worth noting that, even if modifications are introduced in the behavioral description of the application circuits, the VHDL files might be largely modified. Besides, this technique could be excessively rigid: since the state management circuit is created before segmentation, too few I/O pins might be left available to the application circuits.

A more flexible technique consists of counting – during the segmentation process – the memory elements to be managed and dynamically estimating the segment size. Once the segments are found, one auxiliary circuit can be created for each of them separately, while fulfilling all constraints. Since segmentation is performed onto the VHDL representation of circuits, both the segments and the state management circuits are expressed as VHDL components. Merging each pair of these circuits into one implies many changes in both of them. However, by following some simple design rules to write the VHDL description of the circuit components, changes required by the auxiliary circuit are highly reduced. In particular, in the interface of each component we can introduce one signal for each internal memory element; the auxiliary circuit can be built separately and, then, merged with the corresponding segment by simply using VHDL structural style (Figure 1).

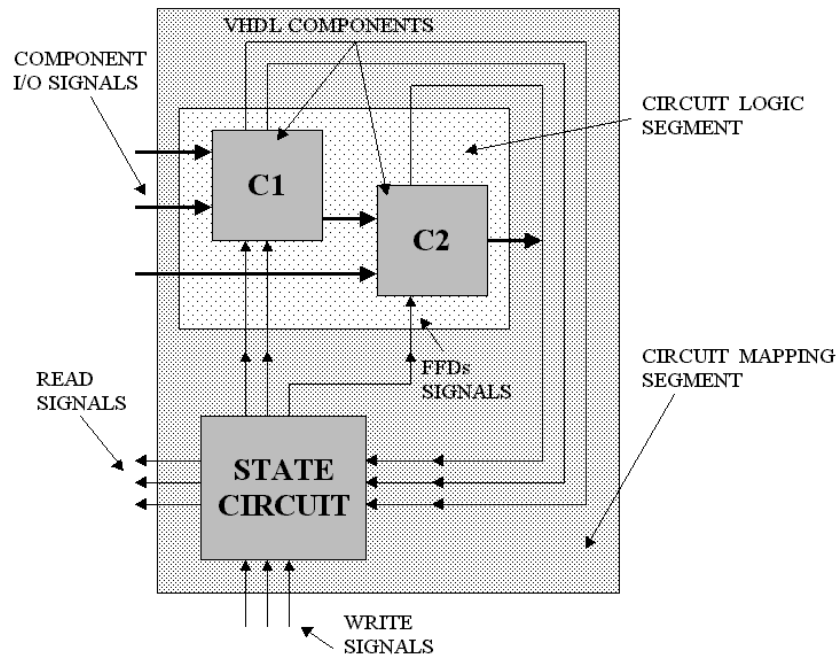


Figure 1: The state management circuit.

Flexibility of this solution allows for several tradeoffs between the number of input/output pins used by the auxiliary circuit and the circuit execution time. In fact, by using less I/O resources for the state management than the number of state bits, the state management will be performed over several clock cycles, but some I/O resources are now available for the application circuit. All problems discussed above can be easily avoided by using an FPGA device which allows to access directly to state information from outside. One of these devices is the Xilinx XC6200 family [18], that we adopted for our experiments. No auxiliary circuit for state management is necessary. Memory elements holding the state in the FPGA are directly read or written by the processor through the I/O resources of the FPGA, by means of the appropriate control signals. In several FPGA devices (including the X6200 family), only one CLB column can be accessed at a time.

2.2 I/O data management

Typically, only one segment per application process is loaded into the FPGA at a time. We must therefore provide suited mechanisms for transferring data from one segment to another in order to accomplish the whole application circuit computation. Output data must be saved automatically at the end of the segment computation and, then, delivered as input to the subsequent segments.

A first solution consists of adopting a dedicated circuit for data storing. This circuit is composed by FFD memory elements available in the FPGA itself and connections between these memory elements and the corresponding I/O pins of each segment loaded in the FPGA.

Drawbacks pointed out in Section 2.1 for state management are still present. In fact, this circuit must be permanently loaded into the FPGA, introducing new problems for operation mapping and signal routing: the resources assigned to the storage circuit cannot be used by other segments, thus creating new constraints for segment mapping. Besides, connecting the internal FFDs to the I/O pins of a segment constraints the mapping of all the other segments that use these data. The high number of segments and signals makes very difficult to satisfy every constraint.

Further problems occur during segment execution. The required persistence of the storage circuits does not allow to share the FPGA in time among different application processes. Besides, the FPGA cannot be reset between different loading since any information contained in the storage circuit would be erased too. Consequently, the content of every CLB used by the currently loaded circuit must be explicitly and individually erased by the operating system to avoid any interference with the following segment.

If the FPGA architecture allows for direct reading and writing the memory elements, a feasible, flexible, but potentially slow solution consists of saving the data in the central memory of the computing system through the operating system.

The above problems concerning data management occur also when sequential circuits are to be mapped on the FPGA. For these circuits is important to analyze the effect of cycles on segmentation and their impact on the performance of segmented circuits. The main performance loss is due to the time spent by the applications in configuring the FPGA configuration RAM. Besides, in general each segment must be executed for only one clock cycle to reproduce correctly the computation of the sequential circuit: at each clock cycle it is in fact necessary to feed back some intermediate result to update the FFDs.

However, whenever a segment is without cycles, it can be executed for several clock cycles, provided that each intermediate result is saved and, then, properly delivered at each clock step. This allows to reduce the number of configuration loading. On the other hand, the presence of cycles encompassing several segments makes impossible to reuse the segment for several cycles. For all the above reasons, design guidelines must be studied to deal efficiently with cyclic dependencies.

3. Design issues for effective segmentation

Segmenting an application circuit for virtual FPGA mapping is similar to the general circuit partitioning problem, which is typically addressed whenever a large circuit need either to be realized in several chips or to be mapped on several FPGA devices. For this problem, many partitioning algorithm are available in the literature (e.g., KLFM [5], FBB [19]). However, these algorithms consider all partitions present in the hardware and accessible at the same time (for instance, in a multi-FPGA system), by having one partition per

device. Their main objective is therefore to minimize the “cut-set” (i.e., the number of interconnections among partitions) in order to maximize the global performance, while satisfying the constraints on the available resources within each device. FPGA configuration is loaded only once per partition at the beginning of the first execution of the circuit.

In segmentation for virtual FPGA, there is only one physical device and the system must be designed to operate also if only one segment at a time is loaded in the FPGA. The main loss of performance is thus due to the time spent in loading the different configurations whenever a new segment need to be executed. It is worth noting that if some segments are needed several times to complete one circuit execution (e.g., in sequential circuits), their configuration might be loaded several times. Since a sequential circuit cycles among its components, cycles among different FPGA segments must be eliminated, e.g., by clustering all components belonging to a cycle into a single segment.

The direct use of conventional partitioning algorithms is therefore not suited since they do not consider the above issues: some modifications are required to adapt them to the specific need of FPGA segmentation.

Partitioning of the application circuit into segments is performed onto the VHDL representation of the circuit itself. The goal of partitioning is to cluster the circuit component so that the size of each cluster is sufficiently small to fit into the available FPGA resources, and components which are strongly correlated in the circuit operation belong to the same cluster so that the number of segment configuration loading is minimized. Clustering must also reduce the presence of cycles involving several segments.

Behavior and performance of the segmentation algorithms depend on different factors, according also to the architecture of the physical FPGA. First, the approach adopted to manage the input/output data influences the estimation of the size of the storage auxiliary circuit, if this last has to be introduced. Second, the presence of cycles among VHDL components may lead to a special treatment of the potential cycles. Due to space constraints, in the following of this paper, we will consider only circuits for which cycles across segments are not present; more details about circuits with this kind of cycles can be found in [20].

3.1 Data saving in the system memory

Saving intermediate results in the central memory greatly simplifies the segmentation generation. The whole FPGA (or at least the part of the FPGA assigned to that application process) is available to implement the application segments, without any constraint imposed by the auxiliary circuit. Besides, the absence of cycles among VHDL components across segments allows the designer simplify the partitioning strategy.

Segmentation can be carried out by adopting a “bottom-up” approach, i.e., by expanding the whole VHDL graph and working onto the produced nodes having minimum granularity. Clusters are created by grouping the description achieved by the full hierarchical expansion. This procedure may become quite difficult since it operates on a flattened description of the application circuit and does not make any use of the hierarchical structure of the original VHDL description.

On the contrary, a “top-down” approach consists of progressively expanding the VHDL description graph, one node at a time, and working onto the nodes which have been obtained so far. The algorithm selects a VHDL component that violates the size constraint and expands it until some sub-components satisfy the given size bound. Then, the algorithm tries to cluster such nodes to minimize the number of segments. To maximize the hardware use and to limit the segment reloading, it is desirable that segment size is as close as possible to the maximum allowed.

When the number of nodes is small, this simple approach allows the designer to have a direct control on the segmentation procedure. Besides, this partitioning algorithm operates locally on a small part of the circuit: this allows an accurate and optimizing control, even though local, on clustering (e.g., by adopting an optimization technique like the branch&bound algorithm). For example, consider the circuit presented in Figure 2. It contains three components (A, B, and C) that violate the size constraint. The algorithm selects one of these three nodes (for example the node A) and expands it into its sub-nodes according to the VHDL hierarchy (nodes A1, A2, and A3 in Figure 2B). If the new nodes generated by this expansion do not satisfy the size constraint (e.g., node A3 in Figure 2B), selection and expansion are repeated until some components are found that do not violate the constraints. If there are some components that individually satisfy the size constraint

(e.g., nodes A1 and A2 in Figure 2B), the algorithm clusters as many of them as possible without violating the size constraint (e.g., nodes A1 and A2 in Figure 2C). Clustering produces segments. Existing segments can be augmented with other nodes after expansion of other nodes if the size constraint remains satisfied. After clustering nodes A1 and A2, node A3 is expanded into nodes A31 and A32 (see Figure 2D) and node A31 is clustered with node (A1,A2), as shown in Figure 3E. Clustering is carried out by avoiding to create cycles among segments. After the whole hierarchy of node A has been explored and expanded, the algorithm continues with the other nodes that do not satisfy the size constraint and has not yet been explored.

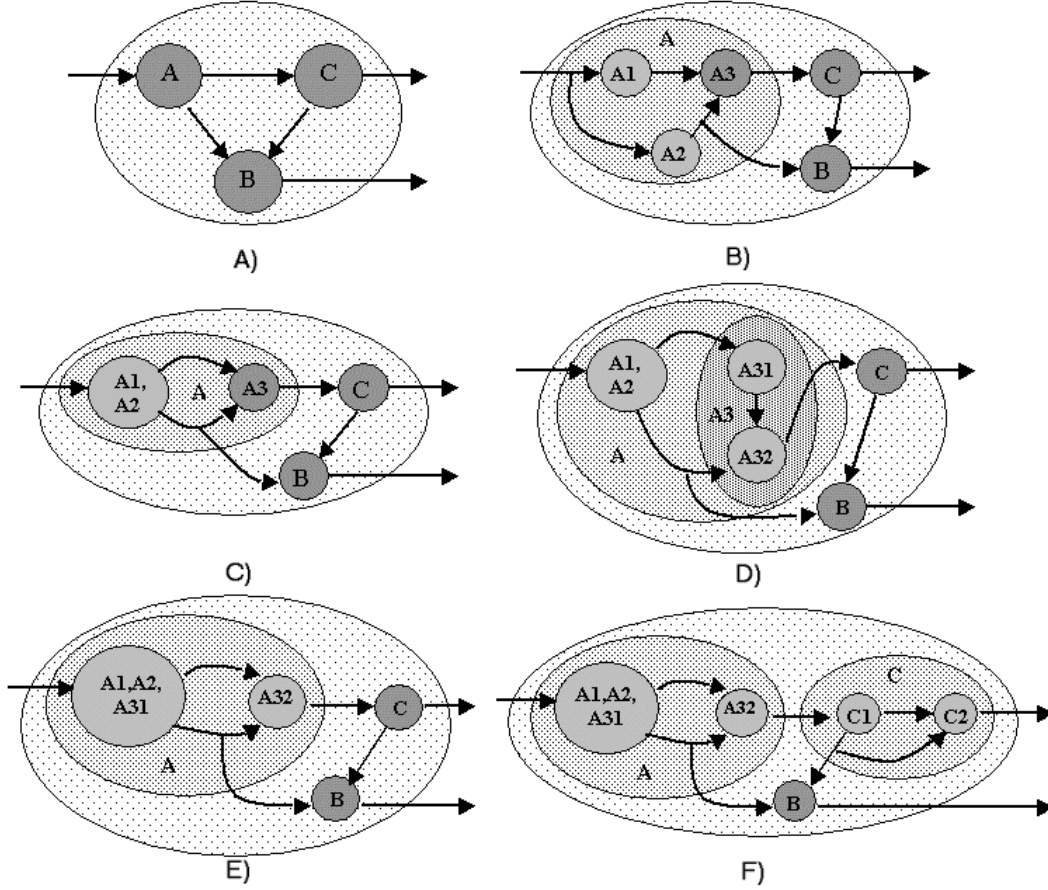


Figure 2: Application of the “top-down” segmentation algorithm.

Since the algorithm has to produce the minimum number of segments by building up partitions having size as close as possible to the size constraint, we need to implement carefully the clustering phase.

A first approach consists of creating only one segment at a time by selecting a *segment base* (i.e., a node in the circuit graph) that will be the core component of the segment. Clustering adds other nodes to this base till the size constraint is satisfied. For instance, let us assume that the components of A and C do not individually violate the size constraint. If the selected base in Figure 2F is node A32, the only clustering actions which are allowed are [(A1, A2, A31), A32] and [A32, C1]. Clustering [C1, C2] is not allowed. In such an approach, only one cluster at a time grows up to the allowed limit by incorporating circuit components: this makes more likely to produce larger segments and, in turn, to minimize their number. It is worth noting that the complexity of the clustering procedure is quite low since only a small part of the circuit is analyzed at a time. The main problem is that this technique does not prevent segments to be much smaller than the maximum allowed.

A second approach was thus adopted to overcome this drawback. The basic idea of hierarchical expansion of the VHDL description is the same of the first algorithm, but it is applied in a different order so as to maximize the size of the resulting segments. Whenever a segment with a too small size is generated, the nodes *near* that

segment (called *compatible nodes*) are expanded until it new components become available to be incorporated into the segment to reach a more suitable size.

The adopted algorithm consists therefore of the following basic steps that are iterated until all circuit nodes have been properly expanded and clustered into segments:

- select a node that violates the size constraint and perform expansion,
- select the segment base,
- select the nodes to be clustered with the current base,
- identify the compatible nodes to expand in order to enlarge the segment.

To select the node to be expanded, we consider the component having the highest execution priority by following the information flow, i.e., the component that must be executed before all the others which have not yet been assigned to a segment. In general, there may be many nodes with same priority. Selection among them could be easily performed randomly, but this may reduce the opportunities for creating large segments.

A better approach to select the node to be expanded consists of analyzing the internal structure of each candidate node, i.e., the relationship among its sub-components: we choose the node producing the sub-partition having the biggest size. This analysis can be done by iterating the algorithm on every node within the candidate node. Even if the number of candidates is usually low, to reduce the computational cost, we limit the application of this algorithm, for instance, by stopping it as soon as components with size within a given range have been obtained. This range can be defined by the designer as a tradeoff between the execution speed and its accuracy. The sub-partitions obtained so far may be reused in the subsequent steps of the algorithm without reprocessing them. Also selection of the segment base can be based on a priority concept which follows the information flow of the circuit. The first segments to be produced are those directly connected to the primary inputs of the circuit, and so on towards the primary outputs.

The approach to perform clustering is critical for an effective segmentation of the circuit. A greedy solution consists of clustering those nodes that yield the greatest gain. However, this often leads to a largely sub-optimal solution. A better technique is provided by the branch&bound optimization algorithm: we will select those aggregations that produce the sub-partition having the largest size. Generally, during each iteration of the clustering algorithm, several nodes are incorporated in the segment. Identification of the nodes to be expanded to augment the segment currently under construction resembles the identification of the initial node to be expanded and, thus, can be solved in a similar way. The complete *NCSM* algorithm for circuit segmentation is thus summarized in Figure 3.

3.2 Data saving into the FPGA

Saving intermediate results into the FPGA itself needs an auxiliary storage circuit that must be permanently loaded in the device.

This imposes a new constraint onto segment mapping. The size constraint becomes equal to the difference between the original size constraint and the size of the new storage circuit. This last quantity depends on the number of signal bits to be managed: therefore, it is unknown before starting the segmentation procedure and, consequently, also the size constraint is unknown before partitioning.

To deal with this problem, we must evaluate the storage circuit size in parallel with segment creation and update it continuously as long as the partitioning proceeds. The evaluation of the storage circuit size is performed as an upper bound by assigning one FPGA cell to each signal bit and allowing the execution of a segment for only one clock step, without any further direct concern to the specific architecture adopted to implement such a circuit. It is worth noting that, later, the specific realization of the circuit and the execution of the segments for multiple clock cycles can be easily included in the analysis. If C is the estimation obtained under the assumption given above, the estimation considering the storage circuit implementation is αC , where α is the number of FPGA cells which are used to manage one signal bit; if n cycles are considered, the estimation becomes $n\alpha C$.

A simple estimation of the complexity of the auxiliary storage circuit consists of counting all signals to be managed and deriving the size of the storage circuit according to the rules given above. This may be a highly poor solution since, generally, a signal needs to be stored only for the interval between its production and its

last use (i.e., the end of the last segment that uses it). A better approach to estimation of the size of the auxiliary storage circuit is the greedy algorithm summarized in Figure 4.

```

// A VHDL graph is expanded until some constraint-compatible nodes are produced.
Procedure Expansion() do
    While(it does not exist any compatible node) do
        Select a node.
        Expands the selected node following VHDL hierarchy.
    End While
End Procedure

// The main algorithm (No Cycles, data Saving in central Memory)
Algorithm NCSM do
    While(the circuit is not optimized) do
        If(Some compatible node exists) do
            Select the current "base".
            Cluster components.
            While(the segment has not a suitable size) do
                If(Some compatible node exists) do
                    Expand a component.
                    Cluster components.
                Else
                    Call Expansion()
                    Cluster components.
                End If
            End While
        Else
            Call Expansion()
        End If
    End While
End Algorithm

```

Figure 3: Circuit partitioning for segmented virtual FPGA with data saved in central memory *and without cycles across segments*.

```

Algorithm Estimation() do
    Insert into list L all components that can be immediately loaded.
    FFD = 0, FREE = 0
    While(L is not empty) do
        For each element of L, compute the
        value  $D = FFD_{PRODUCED} - FFD_{USED}$ 
        Select the component with the smallest
        value D (component P)
        If( $D \geq FREE$ ) do
             $FFD = FFD + D - FREE$ 
             $FREE = 0$ 
        Else
             $FREE = FREE - D$ 
        End If
        Extract P from list L
        Add to L all new graph components that can be
        loaded onto FPGA thanks to P scheduling
    End While
    Return FFD
End Algorithm

```

Figure 4: Estimation of the size of the auxiliary storage circuit.

This algorithm is driven by the parameter Δ , defined as the difference between the number of the required FFD memory elements (i.e., the number of signal bits that must be managed) and the number of freed FFD memory elements (i.e., the number of signal bits “consumed” by the segment and no longer necessary for other segments). The circuit component that must be loaded in the FPGA is thus the one having the smallest value of the parameter Δ . Then, if the number of free and already allocated FFD memory elements is not sufficient to store all the component output bits, additional resources are allocated to the storage circuit. This algorithm does not explicitly assign any FFD to signal bits, but simply gives a heuristic estimation of the number of required FFDs: the designer will be responsible for correctly assigning each FFD during the subsequent phase of segment mapping.

The main drawback of this algorithm is that estimation correctness is strongly dependent on the execution sequence of components. A branch&bound approach could be better used to evaluate the worst-case estimation so that no constraint on the component execution is needed.

Once the designer has chosen how to estimate the size of the storage circuit for his VHDL graph, segmentation can be performed. A “top-down” approach to segmentation is no more feasible since the expansion of a VHDL component may produce new intermediate signals among the sub-components that can influence the storage circuit size. Clustering decisions could be no longer correct after expansion if the size of the auxiliary storage circuit changed.

We adopt therefore a “bottom-up” approach to expand the VHDL graph, similar to the one discussed in Section 3.1, and we perform the estimation of the storage circuit size in parallel. Expansion and circuit evaluation proceeds until all graph nodes respect the size constraint. At the end of this preliminary step, the worst-case of the auxiliary circuit size can be estimated.

Segments can now be produced having a suited size so that they can be loaded in the FPGA with the storage circuit, without exceeding the available FPGA resources. Clustering can be performed by adopting strategies similar to those discussed in Section 3.1. The main difference is that the storage circuit size must be estimated either after each clustering operation or at the end of the whole clustering procedure. In the first case, better results are produced, but the algorithm becomes slower. However, since the size of the auxiliary storage circuit has usually small variations induced by clustering, estimating the size at the end of clustering is acceptable.

Another difference in segment creation concerns the clustering operation. As in Section 3.1, when no more clustering is possible with the current “base”, the algorithm for segment creation selects another base and restarts clustering with the new base to create another segment. The storage circuit size may thus decrease with respect to the expected value. Consequently, further clustering operations could become feasible also with the base previously considered. Clustering should be therefore continued also for the previous segments, leading to parallel segment creation.

New problems arise due to parallel execution of clustering on several bases. For instance, it becomes necessary to choose both which components must be clustered and the base to be used for such a clustering. The best solution consists of adopting a branch&bound approach and considering the effects of every choice on possible clustering with other bases. This solution is not very expensive since there are few alternative choices due to the component size and the relationships among components. The parallel approach can be attractive when restarting clustering on segments considered completed allows to augment them with new components.

4. Concluding remarks and experimental results

This paper introduced the concept of segmentation to realize virtual FPGA for RAM-based devices, in a similar way as operating systems do for virtual memory. This technique allows both for sharing the FPGA device among several application processes and for presenting a high-level view of the physical device with augmented characteristics with respect to the physical one.

The feasibility of the proposed approach have been tested by implementing the software for circuit partitioning and an experimental device driver to included in the operating system. Experiments have been performed on a

personal computer equipped with the AMD K6-2 333-MHz microprocessor, 64MB RAM, and the FPGA board by H.O.T. Works based on the Xilinx XC6216 FPGA device. The experimental environment was developed on the MS Windows/NT 4.0 operating system. The FPGA support introduced in the operating systems consists of a library through which the applications require services to the virtual FPGA and the segmentation device driver handling the segmentation operations (FPGA configuration loading, state management, I/O management, segmentation fault handling). To transfer all data to and from the FPGA board we used the low-level device driver provided by H.O.T. Works and Xilinx.

To measure the performance of the virtual FPGA, we considered an application circuit for signal filtering requiring three physical segments. To verify the influence of multitasking, we run the application circuit alone as well as with other circuits in parallel for other application processes. Table 1 shows the components of execution time for the individual segment as well as for all segments of the application circuit, when segment fault occurs and the operating system needs to load the required ones. The percentage distribution is shown in Figure 5. Measurements depend both on topology of the application circuit, the number and the size of the required segments. The absolute values reported here should be considered only from a qualitative point of view. The percentages give an idea of the relative time spent in each hardware and software component. Besides, the absolute measurements and the percentages may vary with the operating system as well as the hardware architecture of the computer, the FPGA board, and the FPGA device. The reported measurements are therefore specific for the system envisioned, even if the proposed approach can be realized also for other environments.

	3 segments		1 segment	
	mean	variance	mean	variance
Total single computation	315,57	32,98	116,90	32,98
Total device driver (segment fault handling, computation handling)	298,01	34,26	99,34	34,26
Library	2,08	0,95	2,08	0,95
Windows/NT I/O Manager	13,91	5,08	13,91	5,08
Segment fault handling (configuration)	239,03	27,98	79,68	26,36
Computation handling (data and state transfer)	58,98	18,32	19,66	18,15
Hardware execution	1,57	1,03	1,57	1,03

Table 1: Execution time for a segmented virtual FPGA.. Columns on the left side give the values measured for the whole 3-segment circuit. Column on the right side give the values measured for the individual segment. All values are in microseconds.

Higher performance could be achieved if the FPGA board architecture is designed to support segmentation directly, by introducing a structure like the memory management unit for virtual memory.

Better performance can be also achieved by introducing the concept of configuration reusing. In several circuits there are in fact parts which are identical and, consequently, have the same configuration when mapped in the FPGA. Identification of these parts is feasible in a quite easy way when a high level description (e.g., VHDL) is used to describe the application circuits. If a segment is used for one of these parts, it can stay in the FPGA for several clock cycles as long as all the application processes needing it are running. This reduces the number of configuration reloading operations and, in turn, the management overhead, resulting in higher system performance.

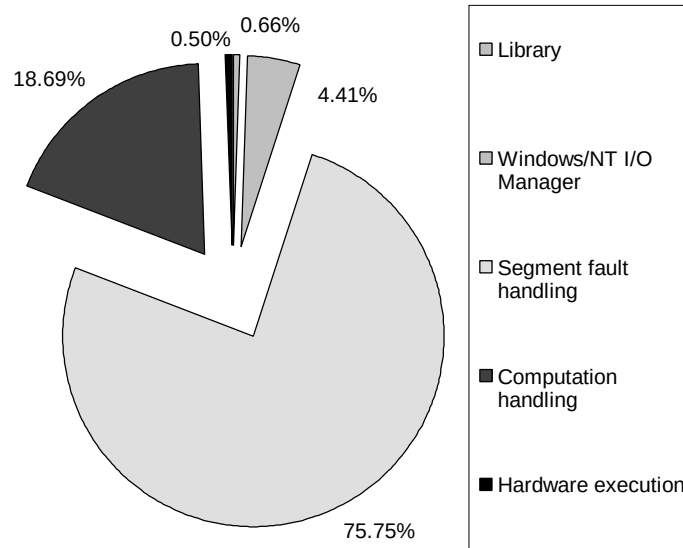


Figure 5: The different components of the execution time.

5. References

- [1] S. Brown and J. Rose, "FPGA and CLPD architectures: a tutorial," *IEEE Design & Test of Computers*, Vol. 13, 1996.
- [2] M.J.S. Smith, *Application-Specific Integrated Circuits*, Addison-Wesley, 1997.
- [3] D.E. Van den Bout, J.N. Morris, D. Thomae, S. Labrozzi, S. Wingo, and D. Hallman, "Anyboard: an FPGA-based, reconfigurable system," *IEEE Design & Test of Computers*, Vol. 9, 1992.
- [4] S. Hauck, "The Roles of FPGAs in Reprogrammable Systems," *Proceedings of the IEEE*, Vol. 86, No. 4, April, 1998, pp. 615-638.
- [5] S. Hauck, *Multi-FPGA Systems*, Ph.D. Thesis, University of Washington, Dept. of C.S.&E., September, 1995.
- [6] D. Chang, M. Marek-Sadowska, "Partitioning Sequential Circuits on Dynamically Reconfigurable FPGAs," *Proc. FPGA 98*, 1998.
- [7] A.S. Tanenbaum, *Modern Operating Systems*, Prentice Hall, 1994.
- [8] W. Fornaciari, V. Piuri, "Virtual FPGAs: some steps behind the physical barriers," *Proc. IEEE Reconfigurable Architectures Workshop*, Orlando, FL, USA, April 1998, pp. 7-12.
- [9] S.M. Scalera, J.R. Vazquez, "The design and implementation of a context switching FPGA," *Proc. IEEE Symposium on FPGAs for Custom Computing Machines*, 1998.
- [10] J.G. Eldredge, B.L. Hutchings, "Run-Time Reconfiguration: A Method for Enhancing the Functional Density of SRAM-Based FPGAs", *Journal of VLSI Signal Processing*, Vol. 12, 1996, pp. 67-86.
- [11] H. Schmit, "Incremental Reconfiguration for Pipelined Applications," *Proc. IEEE Symposium on FPGAs for Custom Computing Machines*, 1997, pp. 47-55.
- [12] J. Granacki, M. Hall, P. Diniz, K. Bondalapati, H. Ziegler, P. Duncan, R. Jain, "DEFACTO: A Design Environment for Adaptive Computing Technology," *Proc. Reconfigurable Architectures Workshop*, Springer-Verlag, 1999.
- [13] T. Bapty, J. Scott, S. Neema, and J. Sztipanovits, "Uniform Execution Environment for Dynamic Reconfiguration," *Proc. Engineering of Computer Based Systems Conference*, March 1999.
- [14] J.S.N. Jean, K. Tomko, V. Yavagal, J. Shah, and R. Cook, "Dynamic Reconfiguration to Support Concurrent Applications," *IEEE Transactions on Computers*, June 1999.
- [15] J. Burns, A. Donlin, J. Hoggs, S. Singh, M. Wit, "A Dynamic Reconfiguration Run-Time System," *FPGAs for Custom Computing Machines*, 1997, pp. 66-75.
- [16] W. Fornaciari, V. Piuri, "General methodologies to virtualize FPGAs in Hw/Sw systems," *Proc. IEEE Midwest Symposium on Circuits and Systems*, Notre Dame, IN, USA, August 1998, pp. 90-93.
- [17] D.L. Perry, *VHDL*, New York: McGraw Hill, 1998.
- [18] *XC6000 Field Programmable Gate Array, XC6000 Data Sheet*, Xilinx Inc., version 1.10, 1997.
- [19] H. H. Yang, D. F. Wong, "Efficient Network Flow Based Min-Cut Balanced Partitioning," *IEEE Transactions on Computer-Aided Design*, Vol. 15, No. 12, December 1996.
- [20] L. Maghini, *FPGA virtuali: la segmentazione*, Ms. Dissertation, Politecnico di Milano, Italy (in italian).