Virtual FPGAs: Some Steps Behind the Physical Barriers

William Fornaciari, Vincenzo Piuri

Department of Electronics and Information, Politecnico di Milano
Piazza L. da Vinci 32, 20133 Milano, Italy
Phone +39-2-2399-3623, Fax +39-2-2399-3411, Email: {fornacia,piuri}@elet.polimi.it

Abstract. Recent advances in FPGA technologies allow to configure the RAM-based FPGA devices in a reduced time as an effective support for real-time applications. The physical dimensions of FPGAs (pinout and gate count) limit the complexity of circuits that can be implemented. In many applications, very large circuits should be realized without requiring either a very large FPGA or many FPGAs; in some real-time systems as well as in multitasking and time-shared environments, it could be valuable to change dynamically the implemented circuit so as to support different applications competing for the FPGA resource. This paper introduces and discusses the concept of Virtual FPGA as an extension of the physical FPGA device: applications have a virtual view of the FPGA that is then mapped on the available physical device by the operating system, in a way similar to the virtual memory.

1 Introduction

The FPGA technologies [1, 2, 3] have been recently developed to provide an efficient computational support for highly demanding applications whenever the use of general purpose microprocessors cannot satisfy performance requirements or the ASIC solution is either far too expensive, or the time-to-market must be short, or the application specification are evolving. Several programmable devices have been developed, e.g., the Programmable Logic Devices (including the Programmable Logic Arrays and the Programmable Array Logic), the Mask-Programmable Gate Arrays, and the Field-Programmable Gate Arrays. Nowadays, FPGA seems the most efficient and effective solution to implement complex circuits with a standard structure that can be easily adapted to the specific application by soft programming based on switch configuration. However, some physical limitations exist in the use of FPGAs. In fact, the circuits that can be implemented with this technology are restricted both in the number of input and outputs and in the circuit complexity (typically, at the moment, FPGAs are available up to 250 K gates and with some hundreds of input and output pins). Even if the largest FPGAs are able to accommodate most of the current largest applications, their cost may be not acceptable in the envisioned market as well as the applications are increasingly demanding for larger devices to incorporate new functionalities, sometimes behind the capabilities of the integration technologies at reasonable costs. On the other hand, in many applications, all circuits realized with the FPGAs are not used all the time: often, some - or even each - of them are used only for a limited time. Therefore, implementing all desired functionalities with dedicated circuits in the FPGAs may lead to a relevant underusing of many parts of the FPGAs themselves. The ideal solution should consists of considering the FPGA as an ensemble of limited physical resources (namely, the input and out pins and the functional blocks) that need to be shared and viewed at a higher abstraction level by the applications running on the computing system containing the FPGA itself. This is the same basic problem occurring in any traditional general-purpose multitasking (possibly time-shared) system to assign the use of shared resources (e.g., processor, memory, input/output devices) in a way that simplifies the use of such resources for each concurrent task as well as provides enhanced features with respect to the ones physically available. In these cases, the operating system [4] is in charge of simulating the high-level view and realizing the dynamic assignment of the resources to the tasks so as to minimize the waiting time preserving the correctness of the operations.

This paper introduces the concept of Virtual FPGA (VFPGA) as an effective and efficient technique to solve the problems mentioned above for the use of FPGAs implementing large circuits or to reduce the costs by adopting smaller FPGAs when the application performance can still be satisfied. The basic idea concerning the FPGA functional blocks reproduce the approaches adopted in the operating systems to support the virtual memory, namely, dynamic loading, partitioning, overlaying, segmentation, and pagination. Similarly, sharing of the input and output pins resembles the management of shared input/output devices.

2 FPGA virtualization

FPGAs are programmable devices composed of a number of functional blocks suitably interconnected [1, 2, 3]. On the market, different kinds of FPGAs are available: symmetrical-array FPGAs (e.g., Xilinx, QuickLogic), row-based FPGAs (e.g., by Actel), Sea-of-Gates FPGAs (e.g., by Algotronix), and hierarchical PLDs (e.g., Altera). The analysis here presented has been focused on the symmetrical-array FPGAs. They are arrays of functional combinatorial logic blocks, possibly containing memory elements to realize sequential functions. Operation of each logic block is selected in look-up tables or in multiplexed devices by suitable setting the corresponding configuration signals. Blocks are interconnected either by using switched busses or multiplexers; long-distance interconnection busses are available to reduce the propagation time in large devices by limiting the number of switches traversed by a signal. The basic results and the approach discussed in this paper can be straightforwardly extended to deal with the other classes.
On the market, are nowadays available also FPGA-based boards for standard personal computers as well as for some workstations: frequently-executed algorithm can be downloaded on these boards to speed up the computation on the main processor. The FPGA board implements a flexible co-processing unit for the standard computing architectures. This introduces the concept of virtual computer: some operations required by the application are mapped on dedicated configurable hardware components (namely, the FPGA) so that the hardware devices of the general-purpose computer becomes specialized. A higher-abstraction level could be envisioned by realizing a computing system composed only of FPGA-based boards so that the whole system operation can be virtualized and downloaded at the beginning of the activities.
In this paper, we afford the problems related to the physical limits of the FPGA devices, namely the number of input and output pins and the number of functional logic blocks. Our goal is to create a Virtual FPGA, i.e., a virtual device having abstract and better characteristics from the point of view of the applications than the physical device. As operating systems [4] do with the virtual memory of general-purpose time-shared multitasking computing systems, the FPGA is virtualized by multiplexing its physical components for all application tasks that need to realize part of their computation in hardware by downloading the corresponding configuration on the FPGA itself. The FPGA can be therefore treated as any other shared hardware resource in the general-purpose multitasking system. Besides, to simplify the application programming, each task concurrently running on the host computer should view the whole FPGA as completely dedicated to itself. In this way, the necessary synchronization among tasks to use the shared FPGA do not need to be included in each application task, but are managed and enforced directly by the operating system as it is accomplished for all other shared resources. The application designer and programmer can therefore focus only on the application aspects of the specific task they are working on, without taking care of all other tasks as well as problems not related to the application but only to the architecture running the application itself.
In brief, for the FPGA components, we are directed to achieve the two typical goals of any operating system for any shared resource: a high-level abstract view of the hardware component (usually with better features than the physical one) and a virtual view of the hardware component itself as completely dedicated to each individual application task.
This paper introduces some basic concepts, which are well-known in the literature on operating systems, into the FPGA-based systems. In particular, the following strategies will be pursued to realize VFPGAs:

- *dynamic loading* is directed to load the FPGA configuration as required by the running application task, either explicitly upon system call or implicitly when the task is started or reactivated by the operating system;

- *partitioning* divides the functional logic blocks of the FPGA in groups so that the operating system can download, independently, a configuration in each of them as required by the corresponding groups of tasks;
- *overlaying* configures part of the FPGA to compute common functions which are frequently used, while the remaining part is used to download specific functions which are typically rarely used or mutually exclusive;
- *segmentation* decomposes the function to be downloaded in the FPGA into smaller parts computing a self-contained sub-function and, as a consequence, having variable size;
- *pagination* partitions the function to be downloaded into smaller portions of fixed size;
- *input and output multiplexing* is used to assign the current inputs and outputs to the logical function associated to the running task or to increase the number of inputs and outputs when there are not enough physically available.

Feasibility of the Virtual FPGA and their effective use in real applications is strictly related to the configuration time. To implement VFPGA, we must consider therefore only the FPGA architectures in which configuration is obtained by programming the interconnection switches or multiplexers in a static RAM (e.g., Xilinx, Altera, and AT&T). Also in this case, the efficiency of the approach when frequent reconfiguration is envisioned is limited by the configuration mode and time, i.e., by the way and time the configuration can be written in the RAM. For example, in the Xilinx X4000 FPGAs, the configuration can be downloaded only serially and completely in no more than 200 ms. Therefore, programmability is restricted in the practice to initial configuration or occasional reconfiguration. In some Xilinx FPGAs families, the connectivity is partially reconfigurable. In these cases, frequent reprogramming of the FPGA is feasible since it may take only a limited percentage of the time dedicated to the other system activities.

In the following, the different virtualization techniques are introduced by taking into account the practical usability as well as the increase of functionalities and performance which provide to the applications competing for the FPGA support. Due to space limitation, only dynamic loading and partitioning are addressed in this paper, a wider analysis can be found in [5].

3 Dynamic loading

In multitasking systems, concurrent tasks may need to use the FPGA to perform specific (usually, independent and unrelated) algorithms in hardware so as to achieve the performance required by the corresponding applications. In some cases, an application may benefit from the speed-up granted by the FPGA execution of different independent algorithms at different points of the task itself. In other cases, it is interesting to be able to run a service algorithm in hardware for all tasks in the system, by selecting the desired algorithm among a given pool; this is typically the case of a device driver when different management options are available (e.g., encoding/decoding, compression/decompression, networking devices).

The common requirement of the above cases is the ability of selecting an algorithm and downloading the corresponding definition on the FPGA whenever the application running on the processor needs it. If the FPGA is large enough to accommodate contemporaneously all circuits required by all applications, a trivial solution is to merge all circuits into only one: each task will use the part of the merged circuit in which it is interested and ignore all other outputs.

The general solution is indeed *dynamic loading* the desired configuration in the FPGA. The basic idea consists of modifying the FPGA configuration as required by the concurrent tasks. Each task in the system must state the algorithms it likes to have executed in the FPGA and provide the suitable description for the FPGA (in terms of RAM configuration). The task designer must take into account these requirements in the design steps and provide the necessary information to the operating system in order to manage the FPGA devices, exactly in the same way as the operating system does for all the other shared resources. To perform these operations, the configuration desired by the task must be declared and stored in the operating system tables at the beginning of the task life, when the task itself is loaded into the system. This can be accomplished either by means of a specific operating system call or a call to the operating system call "fopen" for files and devices management, provided that the implementation of this latter procedure stores the information about the FPGA configuration in the operating system tables when it is called upon an FPGA device with the configuration specified by the programmer as one of the parameters (equivalent to the open mode for regular files and devices).

As for any other task, when a task needing the FPGA is selected by the operating system to become running during the task scheduling for processor virtualization, the operating system sets up the operating environment as desired by the task itself. The virtual machine on which the task assume to be running is activated or reactivated by the operating system so that it was an hardware machine completely and uniquely dedicated only to such a task and all the other concurrent tasks never exist. For the FPGA, this means that the operating system downloads the desired FPGA configuration into the FPGA RAM, by using the information received at task loading and stored in the operating system tables as discussed above. Then, the operating system can put running the task by updating the processor registers accordingly.
If a multitasking *possibly time-shared* system is envisioned, the control of the computing system cannot be released at any time by the running task to the operating system for task rescheduling until the operation in the FPGA has been completed and the results fully produced; otherwise, intermediate results are lost and the final ones will never be produced. If the FPGA is implementing a combinatorial circuit, this means that the operating system simply needs to wait the complete propagation of the computation along the whole data path within the FPGA. This time can be estimated *a priori* by the compiler of the FPGA configuration by simulation as used as one of the parameters for the initialization procedure of the FPGA device. Alternatively, a suitable service logic circuits can be introduced in the FPGA itself to generate a control signal which becomes active only after the completion of the algorithm mapped on the FPGA; the operating system is thus able to determine the completion of the algorithm in the FPGA by checking the status of this control signal.
Conversely, if the operating system is allowed to interrupt the execution of the algorithm in the FPGA before its completion due to the preemption of the task running on the processor in time-shared systems, it must store all information which are necessary to roll-back the computation in the FPGA from the beginning by presenting the initial data. In the case of FPGA implementing sequential circuits, all the above becomes much more complex and difficult to be dealt with since the final results depend not only on the current inputs, but also on the state of the circuit, i.e., on the historical sequence of inputs. This means that the operating system must store also the information about the state of the algorithm in the FPGA, i.e., the value stored in all memory elements within the sequential circuit mapped on the FPGA. This can be accomplished only if accurate design and implementation both of the algorithm and its mapping on the FPGA are taken into account from the initial design stages. First of all, the internal state of the sequential circuit must be observable (i.e., all memory elements must be readable at the FPGA outputs, either directly or indirectly through a sequence of operations) in order to guarantee that the operating system is able to read and store the current state. Then, the state must be controllable (i.e., there must be a sequence of inputs that stores the desired values in the state memory elements) in order to allow the operating system to restart the computation from the exact point at which it was interrupted. In both cases, the state reading and loading operations should be as simple and fast as possible in order to minimize the reactivation time.
If a single task requires more algorithms are mapped onto the FPGA, a behavior similar to the one discussed above is expected from the operating system. The task needs to specify through an operating system call which is the currently desired configuration each time it likes to change it; otherwise, the most recently configuration used by the task is adopted by the operating system.
If a single algorithm needs to be downloaded in the FPGA for all tasks running on the system, the downloading mechanism is basically the same, but it is not usually performed by an application task since it is more related to the configuration of the overall computing system than to a specific application task or set of tasks. As a consequence, the FPGA configuration downloading must be implemented as the execution of a suitable device driver, that is specified and selected - once for all tasks - in the configuration parameters of the operating system. Even if the FPGA configuration is usually not changed during system operation except upon explicit request of the system manager, we still can refer to this kind of downloading as a dynamic loading since it is not strictly embedded in the operating system itself.
The applicability of dynamic loading is limited by the time required to physically download the FPGA configuration and, possibly, the information about the state of the computation. The last case considered above can always be applied since it is equivalent to a typical initialization procedure executed once. Changing the configuration upon explicit request is feasible if it is

required not too often with respect to the time left to the other application activities, or the time slice in *time-shared* systems.

4 FPGA partitioning

Very frequent dynamic loading of FPGA configuration may become a management activity too time consuming for the operating system. If the circuit to be mapped in the FPGA is large, the configuration time may be not acceptable with respect to the actual use of the circuit itself. Whenever the management overhead is relevant, other approaches should be envisioned or even software programming of the algorithm should be considered.

The more drastic solution to reduce the management overhead induced by multitasking is preventing the shared FPGA use. This resource will be considered non-preemptable, i.e., it cannot be released for subsequent reassignment to other tasks until the task holding it has not completed the algorithm that it required and mapped on such FPGA. Any other task needing an already assigned FPGA will enter in the waiting state as long as the reassignment will not grant the mutually exclusive use of the FPGA itself and move the task back into the ready-to-run state. Parallelism of the execution of application tasks may be greatly reduced, even implicitly forcing the scheduling to a strictly FIFO policy.

If two or more circuits are required by the application tasks and they can be accommodated at the same time in the FPGAs available in the computing system, *partitioning* is an effective technique to reduce the number of loading and, possibly, storing operations and increase the overall time available for computation without impairing the parallelism in a relevant way.

The basic idea consists of dividing the resources of the FPGA into disjoint sets, one for each *partition* of the FPGA. Each partition must be usable, independently from the others, by the operating system to load one of the circuits required by the applications. Each partition corresponds to a group of memory elements in the FPGA configuration RAM, being each element associated to the control line defining either the operation of each functional logic block or the routing direction of each interconnection.

Partitions may have the same or different sizes as well as fixed or variable size. In the first case, partitions are created by the operating system at the initialization by taking the corresponding sizes from system configuration file; creation consists of storing the partition identifier of each memory element in internal tables of the operating system for future downloading of FPGA configurations. This assignment will never be changed unless the system is rebooted with another partition configuration file.

In the case of variable-size partitioning, the boundaries of the partitions can be changed dynamically, as long as this does not affect the execution of the algorithms currently loaded in the FPGA. At system bootstrap, one standard partition is created covering the whole FPGA. Then, new partitions are created upon request of the application tasks by the operating system. At each request, one of the unused partitions having size large enough is selected and split in two parts; a new entry is created into the system table for one of the two partitions and the assignment of memory elements in the two partitions is updated. If no idle splittable partition exists, the request should be delayed and the requiring task suspended until a suitable partition becomes available. A task could remain indefinitely waiting for completing the partitioning request if none of the existing partition is large enough. This can be tolerated if existing partitions remain in use by other tasks: however, this is definitely not acceptable that a task is waiting for enough room in a single partition while such a space may be actually available even if split in more idle existing partitions. In such a case, a garbage-collecting procedure must be introduced to merge - when necessary - the idle existing partitions to create continuous large ones. Relocation on partitions is a time-consuming operation since implies downloading of the circuits implementing such algorithms; therefore, relocation for garbage collection cannot be frequently applied in order to limit the management overhead.

Loading of the circuit, associated to an application algorithm that must be mapped onto the FPGA, can be performed in a way similar to the cases discussed for dynamic loading. When a task requires the FPGA configuration loading, it can also specify the desired partition. If it is idle, the operating system assign the partition to the task; otherwise, the task is suspended until the partition becomes available. In the case of fixed partitions, the tasks can be statically pre-assigned to partitions so that each task does not need to specify the desired one. In general, if the partition

is not specified, the operating system takes care of selecting a suitable one among the currently idle partitions; if none exists, the task is suspended until one becomes available. Eventually, in the case of dynamic partitions, garbage collection is started. Creation of partitions in the dynamic case can be realized without an explicit system call, but implicitly by the loading system call that may invoke the partition creation whenever a too large partition is assigned for downloading.
Then, physical loading of the circuit onto the FPGA is executed exactly as in the case of dynamic loading: the operating system stores the circuit configuration in the FPGA configuration RAM corresponding to the partition itself. Note that the configuration to be downloaded may be not completely independent from the selected partition. In fact, the position and the structure (i.e., the physical location of functional logic blocks, input/output blocks, and interconnections) of the selected partition may impose specific constraints on the use of the individual FPGA resources in the partition (e.g., some interconnection paths cannot be implemented). As a consequence, a particular care must be given in the design of the algorithm to be downloaded and in the generation of the corresponding circuit configuration in order to include these constraints. In the case of fixed partitions, the circuits can be easily created since the size and position of the partition may be known in advance; at most, a recompilation step is required whenever a new partitioning is adopted. Unfortunately, this is not the case of the variable partitions and this problem becomes more critical when garbage collection is taken into account: a solution consists of creating a relocatable circuit to be loaded virtually in any location of the FPGA, if resources provided for its configuration are enough and connectable as specified by the computational data flow graph. However, circuit relocation is more difficult to be formalized and standardized than classical code relocation. An assigned partition remain in use to its task until it is released voluntarily or, as in dynamic loading, the operating system rotates it assignment among tasks, possibly, by preempting it and storing the internal state of the circuit currently mapped.

5 Conclusions

The concept of Virtual FPGA has been introduced to overcome the limits of a physical FPGA, allowing to map larger circuits on smaller FPGAs and, as a consequence, to reduce the cost of using these components by avoiding underused components. Several techniques have been proposed to realize VFPGAs, by resembling the approaches available for virtual memories in operating systems. The proposed strategies can be incorporated in any operating system to manage the FPGA devices in a general-purpose multitasking architecture.
Different applications can be envisioned in the future for the VFPGAs since the cost reduction allows to further expand their market. For example, multimedia systems can benefit from the use of VFPGA implementing different voice and image compression/decompression algorithms in order to accommodate different standards efficiently on a limited-size FPGA. Similarly, in telecommunication, modems, faxes, switching systems, satellites, and cellular phones can adapt their operating mode changing the compression and encoding algorithms according to the partners involved in the communication. Besides, high-performance programmable interfaces for networking and complex disk arrays for high-volume fault-tolerant memory storage can be realized with different protocols and standards activated according to the task running on the processor. In embedded control systems, execution of different non-frequent functions (e.g., periodic system testing and diagnosis as well as tuning of the operating parameters) can benefit from the performance achieved by FPGAs with respect to microprocessors.
Some experiments are in progress to design and develop a prototype system incorporating some of the proposed features. The system is composed of a standard personal computer equipped with a board SIGLA by Virtual Computer Inc. .

References

1. S. Brown and J. Rose, *FPGA and CLPD architectures: a tutorial*, IEEE Design & Test of Computers, vol. 13, 1996.
2. 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.
3. M.J.S. Smith, *Application-Specific Integrated Circuits*, Addison-Wesley, 1997.
4. A.S. Tanenbaum, *Modern Operating Systems*, Prentice Hall, 1992.
5. W. Fornaciari, V. Piuri, *Virtual FPGA*, Politecnico di Milano, D.E.I., Internal Report, 1997.