Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2014, Article ID 827613, 18 pages
http://dx.doi.org/10.1155/2014/827613

Research Article

A Top-Down Optimization Methodology for Mutually Exclusive Applications

Alp Kilic,[1] Zied Marrakchi,[2] and Habib Mehrez[1]

[1] *LIP6, Universite Pierre et Marie Curie, 4 Place Jussieu, 75252 Paris, France*
[2] *Flexras Technologies, 153 Boulevard Anatole France, 93200 Saint-Denis, France*

Correspondence should be addressed to Alp Kilic; kilic.alp@gmail.com

Received 17 May 2013; Revised 9 October 2013; Accepted 22 October 2013; Published 17 February 2014

Academic Editor: Nadia Nedjah

Proliferation of mutually exclusive applications on circuits and the higher cost of silicon make the resource sharing more and more important. The state-of-the-art synthesis tools may often be unsatisfactory. Their efficiency may depend on the hardware description style. Nevertheless, today, different applications in a circuit can be developed by different developers. This paper proposes an efficient method to improve resource sharing between mutually exclusive applications with no dependence on the coding style. It takes the advantage of the possibility of resource sharing as done in FPGA and of predefined multiple functions as in ASIC.

1. Introduction

Today electronic devices contain more and more features due to emergence of new embedded applications like telecom, digital television, and automotive and multimedia applications. These applications require on one hand hardware architectures with higher performances, but on the other hand the same architectures should be as small as possible and meet very tight power consumption constraints.

The interesting point which comes with feature-rich platforms is that lots of the features cannot be executed at the same time. Some of the applications are mutually exclusive. For example, in mobile phones, we cannot listen to the music while talking on the phone. As it can be seen in Figure 1, 2 applications which are mutually exclusive have no common outcomes. Mutually exclusive applications give the possibility of resource sharing among other optimizations. Sharing resources between applications may reduce the total area of the circuit by using less hardware. It should be noted that in some cases it may also lead to area increase.

In order to benefit from resource sharing, mutually exclusive applications can be implemented using different methods. The designer can implement these applications in software which will share the same Central Processing Unit (CPU). This solution will be flexible and low cost. However, especially for computation-intensive applications, performances will be far from an ASIC and may not be sufficient for the requirements. It will not offer low-power consumption. The second way is to share the same FPGA as hardware platform. It would yield better area, power consumption, and speed performance compared to the software solutions. But the drawback of this method is the silicon waste. It is due to the fact that an FPGA contains lots of hardware resources to provide unlimited flexibility which is unnecessary. Thus, in an FPGA, the fact that the limited applications are known in advance and the unlimited flexibility is not needed, is not exploited. Moreover, moving from one application to another comes with a reconfiguration of the FPGA by loading the corresponding bitstream. It, often, takes too long time to switch rapidly the application's architecture in a real-time context. Also, more area is needed to store bitstreams. ASICs are more suitable for high-performance systems, but they are not flexible. Thus to have a good tradeoff between flexibility and performance, multimode systems are proposed. These architectures provide both reconfigurability and efficiency in terms of area, performance, power consumption, and reconfiguration time.

One of the goals of multimode systems is to minimize area by reusing hardware resources effectively among different configurations. Conventional scheduling and binding

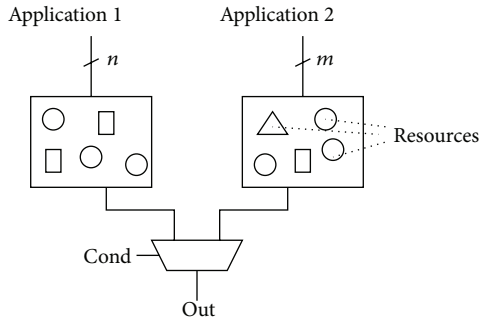


Figure 1: 2 mutually exclusive applications with common resources.

algorithms used in high level synthesis (HLS) can accomplish resource sharing efficiently. The main idea of this work is to propose a new optimization methodology for designing multimode systems. It takes the advantage of the possibility of resource sharing between applications knowing that resources cannot be used at the same time. Unlike HLS which takes algorithmic descriptions as inputs, the starting point of this flow is different RTL specifications which may have been written by different developers or have been generated by an HLS tool. The resulting circuit is called: Multimode ASIC (mASIC).

This paper is organized as follows. Section 2 presents related work around resource sharing on mutually exclusive applications. Section 3 proposes an optimization methodology for multi-mode system design and introduces the mASIC concept. Sections 5, 4, 6 and 7 present in detail the mASIC generation flow and explore different mASIC generation techniques. Then, Section 8 describes the validation process by equivalence checking. Finally, experimental results are presented in Section 9 and we conclude this paper in Section 10.

2. Related Work

One way to design multi-mode systems is to use designers knowledge and experience to identify similar patterns in different modes and to handcraft multi-mode architectures. However this is incompatible with the time to market constraint. Hence multi-mode design needs to be automated. Works on the automation of the design process can be divided into two categories. One uses algorithmic specifications of different configurations to generate an RTL description of a multi-mode system while the second uses RTL descriptions to generate a netlist of it.

In [1, 2], HLS-based approaches to automate the design are proposed. The data flow graphs (DFGs) of the multiple modes are merged into a single graph after each DFG is scheduled separately. Then, the resource binding is performed using the maximal weighted bipartite matching algorithm presented in [3]. In both works, modes are scheduled separately and similarities between configurations are not taken into account. And also, authors did not consider the effect of the proposed methodology on the controller area during the binding step. That is why [4] takes into account the increase of both the controller and the interconnection cost. First it performs a joint scheduling algorithm and then try to optimize the binding step. However, processing the binding step after the scheduling is completed can be penalizing for reducing the interconnection cost. Reference [5] proposes a joint scheduling and binding algorithm based on similarities between datapaths and control steps to limit the extra sharing cost.

Another approach to multi-mode system design is configurable architecture generation. For instance in [6] configurable ASICs (cASICs) are generated for a specific set of benchmarks on the RTL level. Several methods are employed to reduce the number of multiplexers and connecting wires on gate level. cASICs are intended as accelerator in domain-specific systems-on-a-chip. However they are not designed to replace entire ASIC-only chip. cASICs implement only datapath circuits and thus support full-word blocks only. For both the control and data path, [7] proposes an application specific FPGA (ASIF) which is an FPGA with reduced flexibility that can implement a set of applications which will operate at mutually exclusive times. These circuits are efficiently placed and routed on an FPGA to minimize total routing switches required by the architecture. Later all unused routing switches are removed from the FPGA to generate an ASIF. The remaining flexibility is controlled by SRAM cells which are penalizing in terms of area. It uses bitstreams which can take too long time to switch between modes. In addition to the reconfigurable hardware, memories are needed for storing these bitstreams. Time-multiplexed FPGAs increase the capacity of FPGAs by executing different portions of a circuit in a time-multiplexed mode [8, 9]. A large circuit is divided into different subcircuits, and each subcircuit is sequentially executed on a time-multiplexed FPGA. The state information is saved in context registers before a new context runs on FPGA. Tabula [10] commercialized a time-multiplexed FPGA which reconfigures dynamically logic, memory and interconnect at multi-GHz rates with their *Spacetime* compiler. Despite having a multicontext concept, time-multiplexed FPGAs have several drawbacks such as the reconfiguration time overhead and the additional area to store context registers. It does not satisfy demand in multi-mode system design.

This work proposes a different methodology to generate optimized multi-mode architectures. First, RTL descriptions of each modes are synthesized on a given library. Then a multimode ASIC (mASIC) is automatically created using these netlists. mASIC is capable of switching between different modes with a control signal. It contains shared and nonshared resources and inserted multiplexers for shared resources.

3. mASIC Optimization Methodology

mASIC is an automatically created joint netlist that can implement a set of application circuits which will operate at mutually exclusive times. mASIC is generated using mASIC optimization methodology which is a context-aware synthesis method. Applications are synthesized by taking into account the mutually exclusiveness of the applications. This

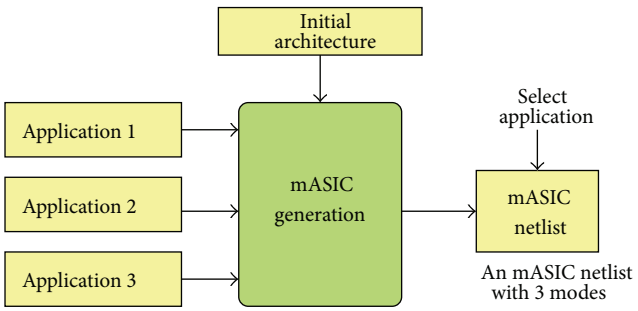


Figure 2: An illustration of mASIC generation concept.

gives to synthesis tool the freedom to share resources between applications.

Figure 2 illustrates the mASIC generation concept. First, an initial architecture that can map any netlist belonging to the given set of mutually exclusive applications is defined. Next, the given netlist is placed and routed with efficient algorithms which favor logical sharing. Efficient placement tries to place the instances of different netlists in such a way that minimum routing switches are required in an FPGA architecture. Consequently, efficient routing increases the probability to connect the driver and receiver instances of these netlists by using the same routing wires. The classical ASIC synthesis flow uses a constructive bottom-up "insertion" approach; the resource sharing is inserted through the addition of multiplexers. This approach prevents the tool to see all applications at the same time to choose the best optimization possibilities. It may be penalizing for sharing logic resources efficiently. mASIC is generated using an iterative top-down "removal" technique. Different applications are mapped onto a given FPGA architecture, and the flexibility is removed from this FPGA to support only the given set of circuits and to reduce its area.

The most important aspect of mASIC is the efficient resource sharing between different application circuits. Resource sharing is done independently from the way of coding the RTL hardware description. This gives the freedom to use any RTL design to generate an mASIC with efficient resource sharing without changing a single line of code. Even though these applications are passed through a software flow which may seem complex, this methodology can be integrated easily into a logic synthesis tool.

The FPGA architecture used for mASIC is the same as the architecture used for ASIF. It is shown in Figure 3. It is a VPR-style (Versatile Place and Route [11]) mesh-based architecture that contains CLBs, I/Os, and hard blocks (HBs) that are arranged on a two-dimensional grid. In order to incorporate HBs in a mesh-based architecture, the size of HBs is quantified with size of the smallest block of the architecture, that is, CLBs. A block is surrounded by a uniform length, single driver unidirectional routing network [12].

4. mASIC Generation Flow

mASIC optimization methodology uses the heterogeneous FPGA environment. The software flow presented in Figure 4

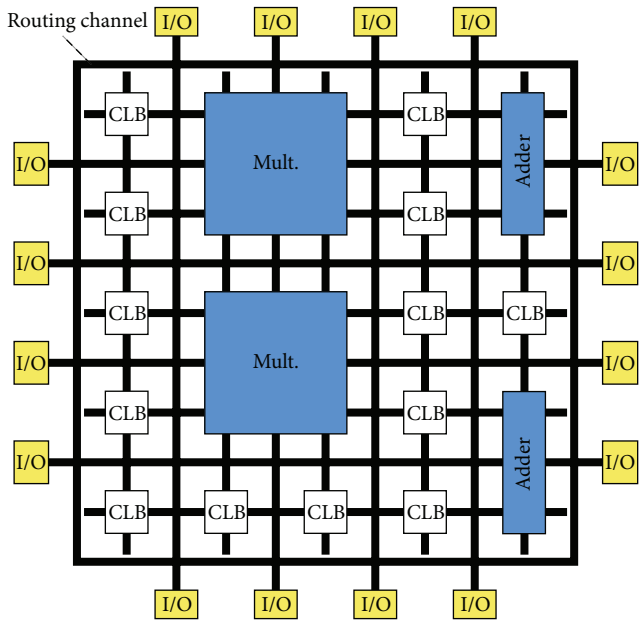


Figure 3: Generalized example of the FPGA architecture.

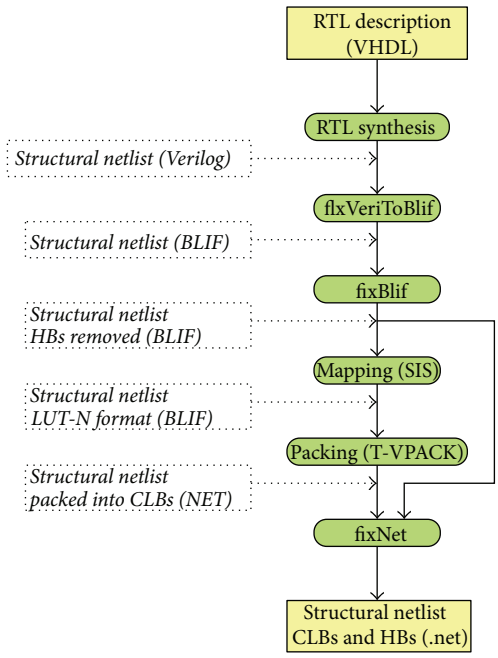


Figure 4: RTL to NET software flow.

transforms RTL descriptions in Verilog or VHDL to their respective netlists in *.net* format, for mapping to the heterogeneous FPGA. This RTL description is synthesized with a logic synthesizer to obtain a structural netlist composed of standard cell library instances and hard block (HB) instances in Verilog. *flxVeriBlif* [13] tool converts the Verilog netlist which contains HBs to BLIF [14] file format. Later *fixBlif* removes all HBs and passes the remaining netlist to *SIS* [15] for technology mapping (synthesis into look-up table format). The size of LUTs is decided in this step. Dependence

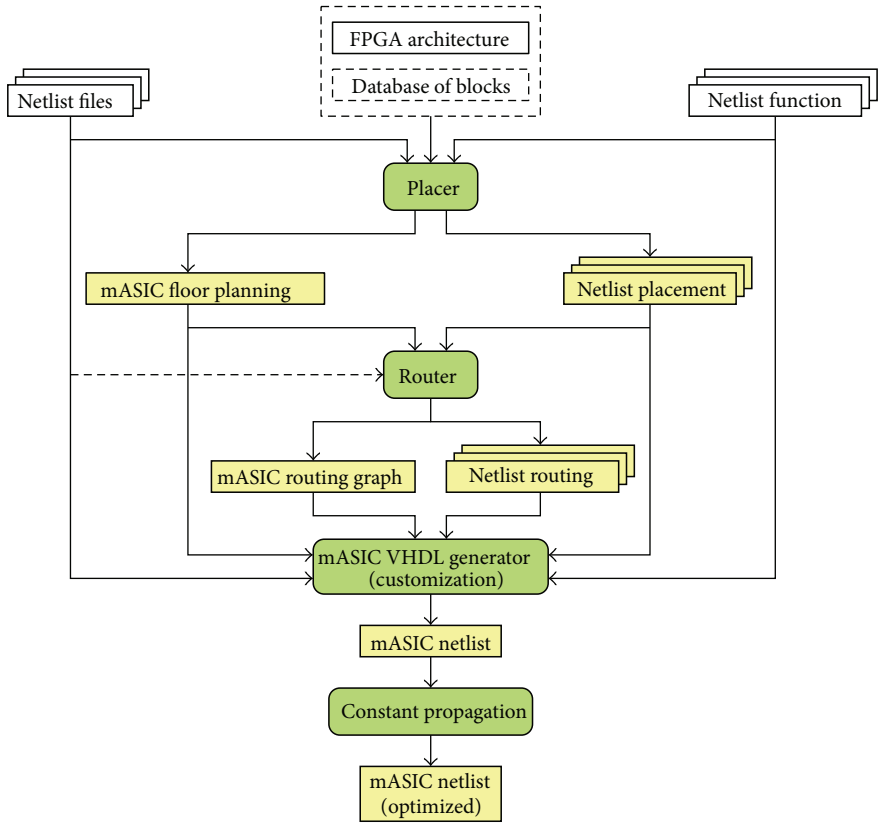


Figure 5: mASIC VHDL generation flow.

between HBs and the remaining netlist is preserved by adding temporary input and output pins to the main netlist. After *SIS*, *T-VPACK* [16], which is a logic packing (clustering) program, packs LUTs and flip-flops together into CLBs. In this work, CLBs contain only one LUT and one flip-flop. *T-VPACK* changes the file format of the netlist from *BLIF* to *.net*. It also generates a function file which contains configuration bits of every CLB in the design. Next, *fixNet* adds all the removed HBs into netlist. It also removes all previously added temporary inputs and outputs. The generated netlist (in *.net* format) includes CLBs, HBs, and IO (inputs and outputs) instances which are interconnected through signals called NETS.

mASIC generation flow is presented in Figure 5. Once the netlists of mutually exclusive applications are converted into *.net* format which contains CLBs and HBs, they are conjointly placed and routed on the target FPGA architecture defined with an enough logic blocks number to handle the given set of netlists. Efficient placement tries to place the instances of different netlists in such a way that minimum routing switches are required in an FPGA. Consequently, efficient routing increases the probability to connect the driver and receiver instances of these netlists by using the same routing wires. Also it favors different netlists to route their nets on an FPGA with maximum common routing paths and tries to minimize the total routing switches required. The placer and the router generate the floor-planning and the routing graph of mASIC. They also generate placement and routing information of each netlist. While routing files hold the information about the configuration on the routing channel, "netlist function" files, generated by *T-VPACK*, have the configuration of each configurable logic blocks. Together, they are called "bitstreams." Each bitstream corresponds to an input netlists.

After placement and routing, mASIC VHDL generator is used to obtain an mASIC netlists. This generator, first, removes all resources, which are not used by any netlists, from the FPGA. Then, this sparse FPGA is customized by removing the remaining flexibility to obtain inflexible multimode ASIC. It is done by removing all the memory points and hard-coding bitstreams through constants and multiplexers. Finally a synthesis tool allows to propagate constants and optimize logic resources and generates an mASIC netlist. Next section gives a brief overview about the placement and routing algorithms used in the mASIC generation flow.

5. Efficient Placement and Efficient Routing

Efficient placement is an internetlist placement optimization technique which can reduce the total number of switches required in an FPGA. It tries to place driver instances of different netlists on a common block position, and their receiver instances on another common block. Later, efficient routing increases the probability to connect the driver and

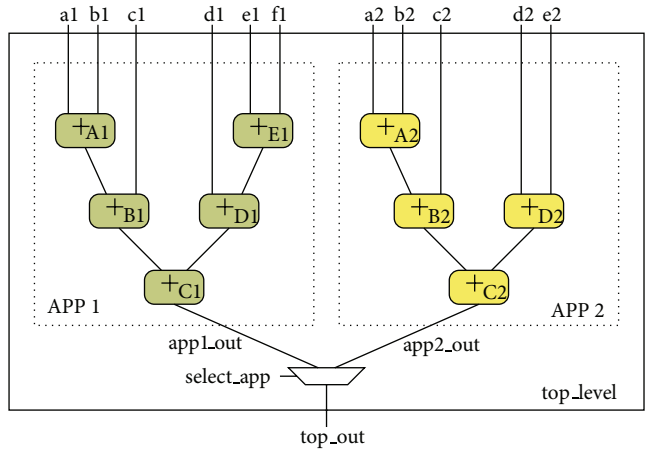
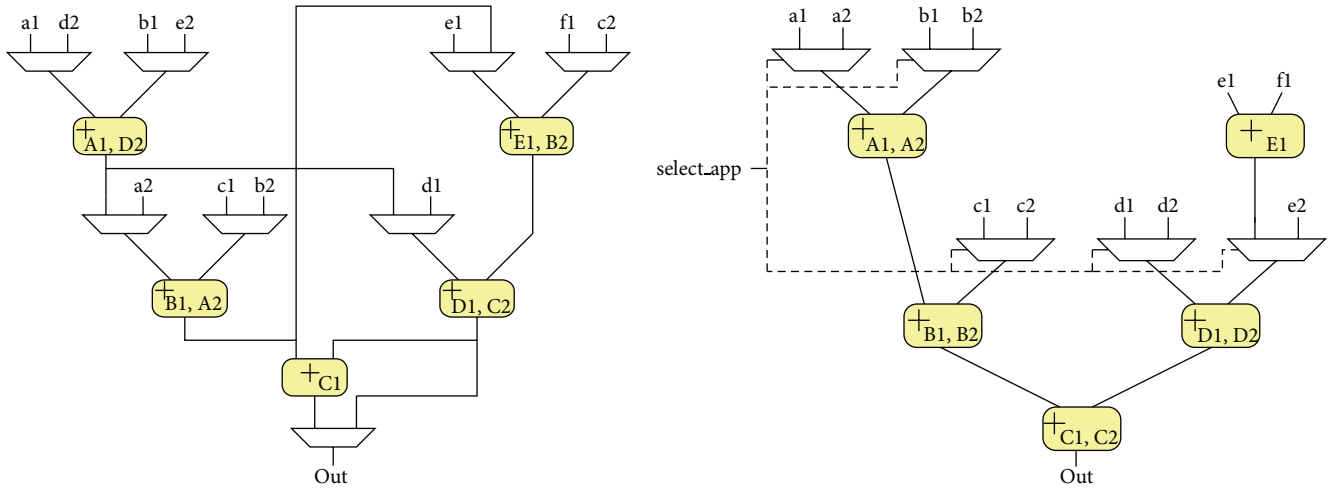


Figure 6: Two mutually exclusive applications: APP1 and APP2.



(a) Placement in common synthesis method

(b) Efficient placement in mASIC optimization methodology

Figure 7: Normal placement versus efficient placement.

receiver instances of these netlists by using the same routing wires.

The advantage of the efficient placement can be understood with the help of an example. Figure 6 describes 2 mutually exclusive applications. The first application (APP1) contains 5 adders while the second (APP2) contains 4. These applications are written in VHDL and are instantiated in a netlist called *top_level*. This netlist, written also in VHDL, has one output which is coming either from APP1 or from APP2. The mutually exclusiveness has been inserted through a multiplexer. When these applications are placed on an architecture which contains 5 adders, multiplexers are inserted in order to share the adders between the applications. The number of multiplexers is related to the placement of the resources which is very important to ensure wire sharing to use less multiplexers. When the efficient placement is used, as shown in Figure 7(b), adders are perfectly paired. Therefore there are only 5 multiplexers to switch from APP1 to APP2. However, the placement which is done with a common synthesis tool is not efficient. As shown in Figure 7(a), the resulting netlist contains 8 multiplexers instead of 5. It is due to the placement of adders. Details regarding the efficient placement algorithm are explained in [7].

After placement of multiple netlists on the predefined architecture, netlists are routed efficiently in order to minimize the required number of switches and routing wires. This is done by maximizing the shared switches required for routing all netlists on the FPGA. The efficient wire sharing encourages different netlists to route their NETS (signals) on the given architecture with maximum common routing paths. It is a top-down routing technique; different applications are mapped onto a given FPGA architecture which contains routing resources, and the flexibility is removed from the FPGA to support only the given set of circuits and to reduce its area. Figure 8 shows an example of the wire sharing done by the efficient routing algorithm. It can be seen that there are different ways to route shared logic blocks. In this example, the efficient wire sharing needs 2 mux-2 to share

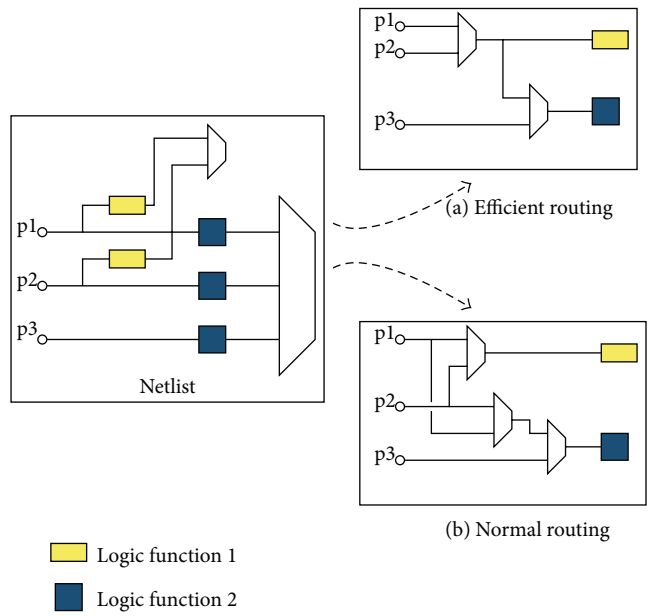
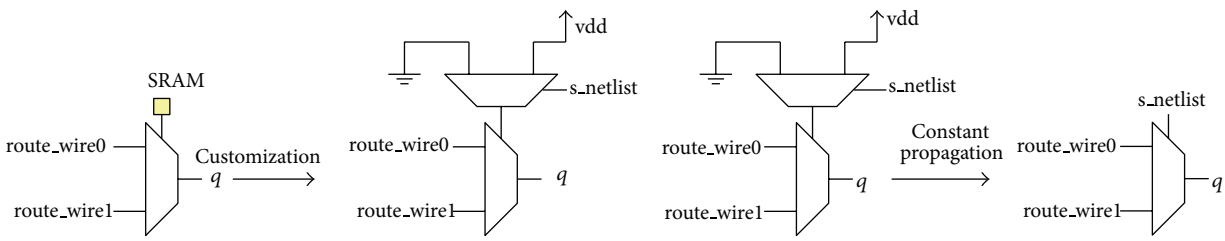


Figure 8: Normal routing versus efficient routing.



(a) Transformation of an SRAM into a multiplexer

(b) Constant propagation on the inserted multiplexer

Figure 9: Hard coding of an SRAM in the routing channel.

Logic Block 1 and *Logic Block 2*. However, a normal routing may use 3 mux-2. At the end, both circuits have the same functionality but the efficient routing uses less multiplexers. As for the efficient placement, further details regarding the efficient routing algorithm are explained in [7].

After placement and routing, mASIC VHDL generator is used to obtain mASIC netlists. This generator first removes all resources, which are not used by any netlists, from the FPGA (initial architecture). Then this netlist which still contains configurable memory points is customized. It is done by removing all the memory points and hard-coding bitstreams generated by the netlist bitstream generator, through constants and multiplexers. Finally, a synthesis tool allows to propagate constants and optimize logic resources and generates an mASIC netlist. Next section describes how bitstreams, which will be hard coded in customization stage, are created.

6. Customization and Constant Propagation

In a traditional FPGA architecture, logic blocks resources occupy less area than routing resources [17]. Since an ASIF is generated by removing unused routing resources of an FPGA, the logic area percentage in an ASIF is higher than an FPGA. It means that the optimization of logic resources may also bring major area advantages.

The initial FPGA architecture used in this work uses SRAM cells, like most FPGAs, to control pass transistors on routing connections, multiplexers, and LUTs on CLBs. When unused resources are removed from the FPGA to obtain a sparse FPGA (ASIF), it still contains SRAMs, thus, limited flexibility. But this flexibility cannot really be exploited since configuration tools cannot guarantee it. Different sets of application netlists, mapped on an ASIF, program the SRAM bits of an LUT differently. In the customization stage, all these remaining memory points are replaced by constants and multiplexers to optimize both logic and routing resources.

Figure 9(a) illustrates the transformation of an SRAM into a multiplexer for 2 different netlists. The SRAM is controlling a multiplexer which introduces the reconfigurability either in the routing channel or in a LUT. As bitstreams of all netlists are known in advance, every memory point can be replaced by a multiplexer that takes hard-coded bitstream as an input. Then the "s_netlist" signal allows to choose which netlist to use.

```
instMux_SRAM_1   :   mux2
PORT MAP(
        cmd    => s_netlist,
        i0     => constant_zero,
        i1     => constant_one,
        q      => cmd_mux_routing_1
);
instMux_Routing_1   :   mux2
PORT MAP (
        cmd    => cmd_mux_routing_1,
        i0     => route_wire0,
        i1     => route_wire1,
        q      => out_mux_routing_1
);
```

Listing 1: An example for the replacement of an SRAM by a multiplexer.

Listing 1 gives an example of the routing channel of an mASIC for 2 applications. It shows the replacement of an SRAM by a multiplexer. "instMux_Routing_1" is a 2-input multiplexer (mux-2) used in the routing channel. The output "out_mux_routing_1" connected either to "value_for_netlist0" or to "value_for_netlist1." In an FPGA this multiplexer is controlled by an SRAM and programmed by the corresponding bitstream. But in mASIC it is controlled by another mux-2: "instMux_SRAM_1." It takes "1" or "0" as inputs and is controlled by an input signal of the circuit: s_netlist. This signal decides which application is going to be active. It can be controlled by the user on the field.

After the customization stage, an intermediate joint netlist, which contains lots of multiplexers and constants, is created. Then, a common synthesis tool (e.g., Cadence RTL Compiler [18]) performs a constant propagation optimization on the input joint netlist. Through this process, the synthesis tool performs logic optimizations on multiplexers inserted in the customization stage. The main goal is to improve the efficiency of the synthesis tool by shaping the input circuit. Later, mASIC can be implemented as an ASIC or in an FPGA.

When the code in Listing 1 is given to a synthesis tool, it will be optimized. Synthesis tool propagates all constants in the circuit. In this particular case, constants in the "instMux_SRAM_1" are propagated. As a result, synthesis tool removes this multiplexer and replaces "cmd_mux_routing_1" by "s_netlist." So the multiplexer which is in the routing channel will be controlled directly by the input of the circuit. This is illustrated in Figure 9(b).

mASIC contains SRAMs also in logic resources (LUTs in CLBs) and these SRAMs have to be customized as well. Figure 10 shows an example of a 3-input Look Up Table (LUT-3) of a multi-mode circuit containing 3 different netlists: N1, N2, and N3. Each netlist has a specific bit vector which can be mapped on 8 SRAM cells of a LUT-3. After the customization, memory points are replaced by multiplexers. Inputs of these multiplexers are hard-coded bitstreams of each netlist. Inserted multiplexers are controlled by the *s_netlist[0:1]* signal. This signal is carried into the circuit interface and it is used to select the desired application. It is the same signal which is used to control the multiplexers inserted in the routing channel.

Figure 11 illustrates the constant propagation optimization on a customized LUT-3. First, constants are propagated through the multiplexers that replace memory points (Figure 11(a)). Then, the propagation continues inside the LUT-3 by replacing multiplexers by logic gates or removing them completely. As shown in Figure 11(b), optimized circuit contains less logic resources than the initial circuit which was a LUT-3. It should be noted that after the customization and the constant propagation stage, the circuit has lost completely its reconfigurability.

7. Reordering of LUT Input Pins

This work tries to shrink the total area of given mutually exclusive netlists. The proposed methods efficiently place and route these netlists on a given FPGA architecture in order to share maximum resources between netlists. Later all unused resources are removed and all memory points are replaced by hard-coded bitstreams.

There are two types of logic resources: hard blocks and CLBs. In this work, hard blocks are considered as combinatorial blocks like adders, multipliers, and so forth. Thus, they have not to be customized for a particular netlist. Each netlist uses the same hard blocks. However, CLBs contain a look-up table (LUT) and a flip-flop. Each netlist may have a different configuration for a LUT. As explained in the previous section, in the customization stage, memory points within LUTs are converted to multiplexers that take configuration bits as inputs. Then these configuration bits are propagated throughout the LUT to optimize the logic. In this perspective, it is important for netlists to have the same configuration bit for the same memory point. As it can be seen in Figure 10, second configuration bit, from the top, is equal to "1" for each netlist (N1, N2, and N3). That is why the second memory point is replaced by "1" which is propagated and

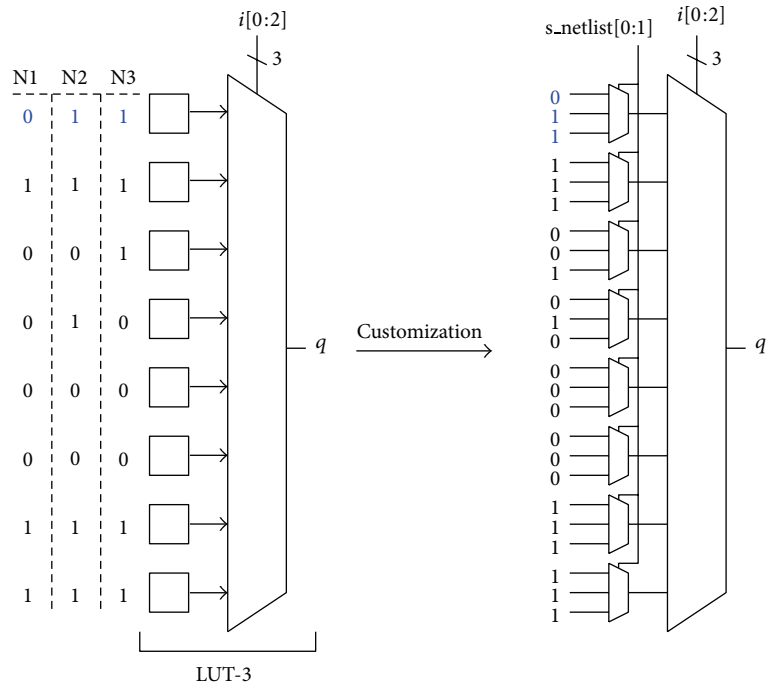


Figure 10: Customization for 3 netlists (N1, N2, and N3).

allows more optimizations on the LUT. The same situation occurs in the 5th, 6th, and 7th bits from the top.

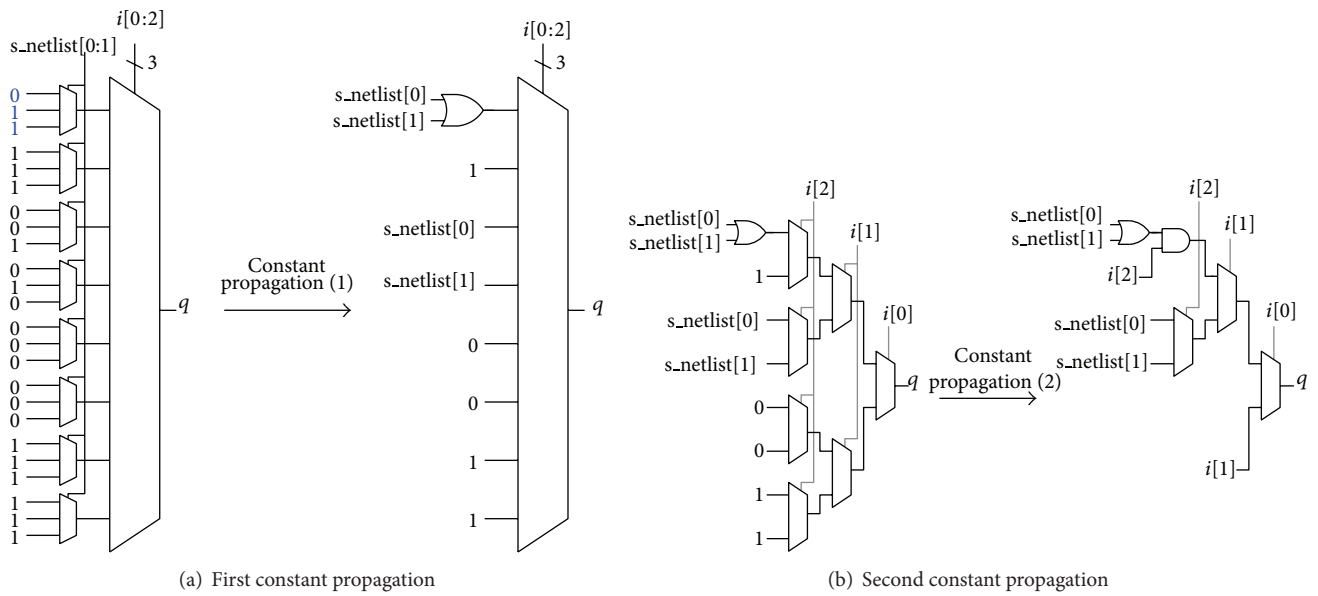
This section presents a method to modify LUT configurations by reordering their input pins. This method serves to find more common bits in different netlist configurations placed on the same LUT. It is done by comparing every possible LUT configurations. By changing its input pins, a netlist can have $n!$ different configurations for a particular LUT, where n is the number of input pins. In total, there are $(n!)^N$ different combinations for customizing a particular LUT where N is the number of netlists using this LUT. The main objective is to find the best combination to get more common configuration bits for all netlists. The resulting combination may introduce more constants instead of multiplexers and allows more optimizations by constant propagation.

Here we give a detailed example for better understanding of the LUT input pin reordering. Suppose 3-input Boolean functions of two different netlists (Netlist-1 and Netlist-2) need to be mapped on the same LUT-3. There are $(3!)^2$ different combinations to customize the LUTs. In this example, in order to simplify the figure, the pin order of the first Boolean function is fixed to default order which is "A-B-C." That is why the LUT configuration of Netlist-1 is never changed. On the other hand, the pin order of the second function takes all the possible values (A-B-C, A-C-B, B-A-C, B-C-A, C-A-B, and C-B-A) to find the most suitable configuration. For each configuration pair, the LUT is customized and optimized as described in Section 6. Figure 10 shows the customization for default pin order. For each pin order, the LUT is customized in the same way. The constant propagation process is illustrated in Figure 12. Optimized LUTs are shown on the right of the configuration pairs.

Figure 12(a) presents the default pin order without reordering. Pin orders of both netlists are fixed to "A-B-C" and there are 4 common bits. A different pin order for Netlist-2 can increase or decrease the number of common bits. It seems when the LUT configuration of Netlist-2 is changed according to pin order C-A-B, it becomes a perfect match for the LUT configuration of Netlist-1 (Figure 12(e)). Consequently, after customization and constant propagation, this order allows to have the smallest area among all combinations.

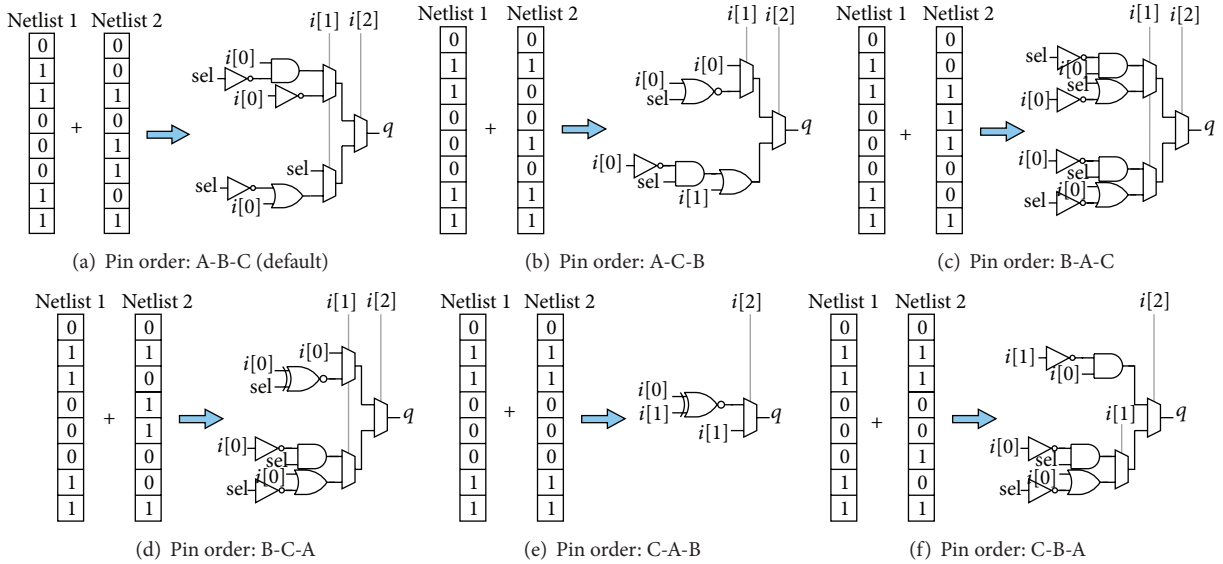
mASIC generation flow (Figure 5) is extended in order to support the LUT optimization. Once all netlists are placed, the best pin order for each LUT in each netlist is explored. Then, netlists and netlist functions are regenerated regarding new pin orders which allow more constant propagation optimizations. Later these new files are used for routing and VHDL generation. The modified mASIC generation flow is presented in Figure 13.

The drawback of this method is the increased routing area. Normally, routing congestion can be decreased if a net or signal is allowed to route to the nearest LUT input, rather than to the exact LUT input as defined in the netlist file. As the congestion decreases, the number of multiplexers in the routing channel decreases. If the router does not use the default pin order to avoid using a multiplexer, later, the LUT configuration will be changed according to the new routing. But when using LUT input pin reordering method, the order of input bits and thus the configuration of look-up tables can be changed before routing in order to find common bits



(a) First constant propagation (b) Second constant propagation

Figure 11: A constant propagation example for 3 netlists (N1, N2, and N3).



(a) Pin order: A-B-C (default) (b) Pin order: A-C-B (c) Pin order: B-A-C

(d) Pin order: B-C-A (e) Pin order: C-A-B (f) Pin order: C-B-A

Figure 12: LUT input pin reordering example.

between netlists. Once changed, they cannot be rechanged by the router. Otherwise this stage becomes useless. The fact that the router cannot modify the order of input pins may increase the number of multiplexers in the routing channel.

An algorithm is required to find the best pin orders among permutations. Such an algorithm is shown in Algorithm 1. First, all sites declared in the architecture file are checked. If a site contains a LUT which is used by multiple netlists, then it can be optimized. In this case, for all possible permutations of the pin order of all netlists, common bits between netlists are counted. The permutation which provides the maximum number of common bits gives the best pin orders for all netlists. As it can be seen in Line 12 of the algorithm, in each permutation, LUT configurations are changed by using ChangePinOrder() function. Listing 2 shows an example for changing LUT-3 configuration of the Boolean function (F) when LUT pin connectivity changed from ABC to BCA. It shows that the configuration bits in a LUT are considered as an array. A new LUT configuration is computed from the original LUT configuration by simply swapping values according to different pin orderings. There are 6 different orderings of pins for a LUT-3. To support also other LUT sizes, a generic algorithm needs to be written that can automatically change the configuration information of any LUT size using any pin ordering. Such a generic algorithm is shown in Algorithm 2. Line 5 calls the "RecursiveLoop" function to compute the new LUT configuration for the new pin order.

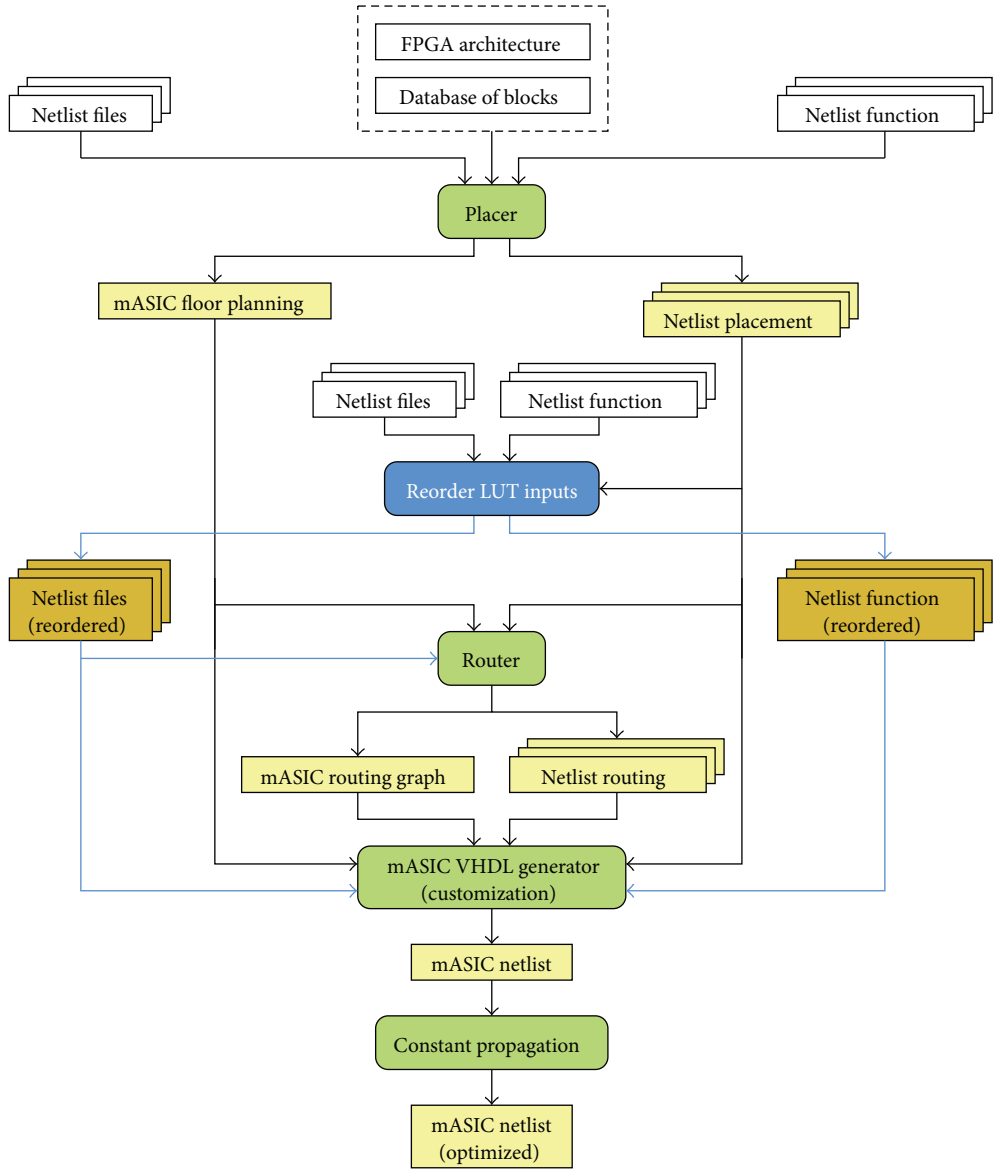


Figure 13: mASIC generation with LUT optimization.

Figure 14 gives more details about how to compute all possible permutations of the pin order of all netlists. In this example 3-input Look-Up Tables are used. That is why for 1 netlist there are 3! and for N netlists there are $(3!)^N$ possible permutations. A randomly chosen permutation is highlighted in the figure: $\text{Net1}_{\text{CAB}}, \text{Net2}_{\text{BCA}}, \ldots, \text{Net}N_{\text{ACB}}$.

8. Validation

mASIC generation flow contains many different steps. It is crucial to validate the functionality of the generated netlist to avoid errors which can be introduced in the flow. Simulation techniques at various design levels are widely used for the verification of designs. They compute the output values for given input patterns using simulation models. Because the quality of verification deeply depends on the given input patterns, it is possible that there could be design bugs that cannot be identified during simulations. In addition to this, simulation is a slow process. That is why formal verification techniques have been researched and developed.

Formal verification techniques ensure 100% functional correctness and they are more reliable and cost effective, less time consuming. The main concept of this technique is not to simulate some vectors and instead prove the functional correctness of a design. In a formal verification, specification and design are translated into mathematical models. It explores all possible cases in the generated mathematical models to validate the circuit. There are different proof methodologies employed, but the methodology used in this work is "Equivalence Checking." The goal is to ensure the equivalence of two given circuit descriptions between different levels of abstraction.

```
(1)  for all SITES in the architecture do
(2)      validNetlists ← Netlists using this SITE
(3)      if (SITE contains a CLB) and (# of validNetlists > 1) then
(4)          eqBitsMax ← 0
(5)          for all Permutations do
(6)              eqBits ← Count Equal Bits in LUT Configurations of validNetlists
(7)              if eqBitsMax < eqBits then
(8)                  eqBitsMax ← eqBits
(9)                  bestPermutation_tmp ← currentPermutation
(10)             end if
(11)             for all validNetlists do
(12)                 ChangePinOrder(currentPermutation)
(13)                 ChangeConfigTable(validNetlist)
(14)             end for
(15)         end for
(16)         bestPermutation ← bestPermutation_tmp
(17)         for all validNetlists do
(18)             ChangePinOrder(bestPermutation)
(19)             ChangeConfigTable(validNetlist)
(20)         end for
(21)     end if
(22) end for
(23) Regenerate NETLIST and FUNCTION files.
```

Algorithm 1: Algorithm for the LUT optimization.

```
for A = 0 to 1
  for B = 0 to 1
    for C = 0 to 1
      NewLUT[Bx4 + Cx2 + A] = OldLUT [Ax8 + Bx4 + Cx2]
(i) When order of pin connectivity with LUT-3 changes from ABC to BCA
(ii) NewLUT and OldLUT have elements
```

Listing 2: LUT configuration swapping.

```
(1)  for all CLB instances do
(2)      ConfigInfo ← Original configuration information for a LUT instance
(3)      pinOrderInfo ← Compute new pin ordering (default pin ordering is 0, 1, 2, 3,...)
(4)      TotalInputPins ← Total input pins of a LUT
(5)      RecursiveLoop(0, TotalInputPins)
(6)      ConfigInfo ← newConfigInfo
(7)  end for
(8)  function RecursiveLoop(bit, pinNum)
(9)      pinIndex[pinNum] ← bit * (2^pinNum)
(10)     newPinIndex[pinNum] ← bit * (2^pinOrderInfo[pinNum])
(11)     if pinNum = 0 then
(12)         index ← sum of all entries in pinIndex
(13)         newIndex ← sum of all entries in newPinIndex
(14)         newConfigInfo[newIndex] ← ConfigInfo[index]return
(15)     end if
(16)     for bit = 0 to 1do
(17)         RecursiveLoop (bit, pinNum − 1)
(18)     end for
(19) end function
```

Algorithm 2: Algorithm to change LUT configuration for different pin connectivity order.

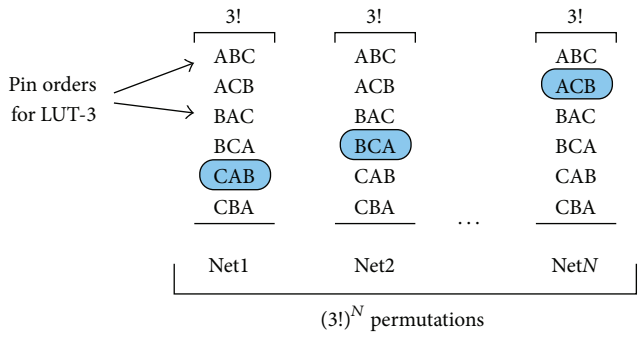


Figure 14: Possible permutations for N netlists using LUT-3.

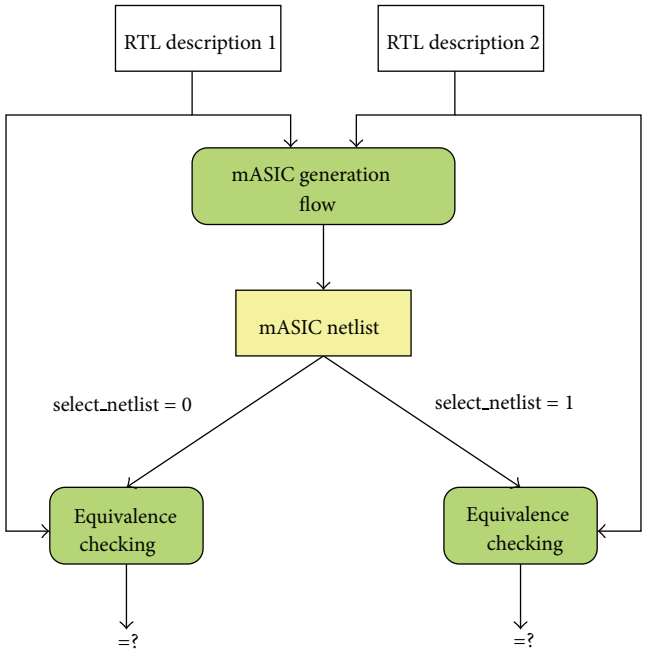


Figure 15: Equivalence checking flow for mASIC.

In order to test the functionality of the generated architecture, we use the validation flow shown in Figure 15. First, RTL descriptions (in VHDL or Verilog format) of mutually exclusive applications are given to the mASIC generation flow in order to obtain an mASIC which can run one application at a time. Then each different configuration of mASIC is compared with its RTL description by using the equivalence checking method.

mASIC can be configured to the corresponding RTL descriptions by varying "s_netlist" input. When "s_netlist" is forced to 0, mASIC can be compared with the first RTL description; when "s_netlist" is forced to 1, mASIC can be compared with the second RTL description and so on. In this work, Cadence Conformal Logic Equivalence Check (LEC) [18] is used as an equivalence checker for the validation of mASIC. Experiences show that the functionality of applications does not change throughout the mASIC generation flow.

Table 1: MCNC benchmark circuits [19].

Index	Netlist name	Number of CLBs			
		LUT-2	LUT-3	LUT-4	LUT-5
1	spla	5671	3772	2592	2332
2	diffeq	3046	1953	759	693
3	apex4	1763	1176	873	755
4	ex5p	1120	727	572	505
5	tseng	4244	2362	1052	949
6	apex2	2977	2000	1017	1025
7	seq	2955	1917	1126	1031
8	ex1010	3020	2003	1404	845
9	alu4	2446	1567	805	722
10	misex3	2138	1420	843	703

9. Experimental Results and Analysis

To evaluate the efficiency of the proposed mASIC generation flow, we use homogeneous (only CLBs based) and heterogeneous benchmark netlists. For both architecture types, we generate mASICs which contain up to 5 netlists (mASIC-2 contains 2 netlists, mASIC-3 contains 3 netlists, and so on). First, we explore the effect of the LUT size by applying mASIC generation techniques on a set of MCNC designs (Microelectronics Center of North Carolina designs) [19]. As presented in Table 1, these benchmark netlists do not contain hard blocks and all of their logic resources are implemented in CLBs with different LUT sizes. It should be noted that a CLB contains 1 LUT. Later, we apply the LUT input pin reordering method presented in Section 7, on MCNC designs to evaluate the impact on area. Finally, we use OpenCores [20] netlists which contain different types of hard blocks to compare mASIC optimization with the common synthesis method. OpenCores benchmarks are shown in Table 2. There are 2 SETs of heterogeneous netlists. While SET1 combines different applications, SET2 contains different configurations of a single application: FIR filter.

In this work, we use the common synthesis method to compare the results with mASIC. Both methods are illustrated in Figure 16. In the common ASIC synthesis method, RTL descriptions of digital base bands are encapsulated in a top level. Then their outputs are connected to a multiplexer in order to choose which standard is used at that moment. This new top level is shown in the right branch of the Figure 16. Finally the RTL description of this configurable digital baseband is synthesized with Cadence RTL Compiler. A 130 nm standard cell library is used during synthesis.

9.1. LUT Size Effect on mASIC. According to [17], for an FPGA, LUT sizes of 4 and 5 are the most area efficient for all cluster size. As for an ASIF, [21] claims that LUT-2 and LUT-3 provide the best results in terms of area. This difference is due to the fact that as LUT size increases, the amount of global routing resources is reduced as more NETS are completely absorbed and implemented by the local interconnect inside LUTs. In an FPGA, the routing network occupies 80–90% area whereas the logic area occupies only

Table 2: OpenCores heterogeneous benchmark circuits [20].

SET	Index	Netlist name	Adder (total)	Mult. (total)	LUT-2	LUT-3	LUT-4	Function
1	1	diffeq_c_systemC	4	3	1356	699	473	Diff. equation solver
	2	cf_fir_16_16_16	16	17	1492	1492	1492	16th order FIR filter
	3	fm_receiver	18	1	2463	1766	1583	FM receiver
	4	lms	5	10	1143	971	946	Adaptive equalizer routine
	5	rs_encoder	15	16	200	200	200	Reed Solomon encoder
2	1	cf_fir_24_16_16	24	25	2114	2114	2114	24th order FIR filter
	2	cf_fir_16_16_16	16	17	1492	1492	1492	16th order FIR filter
	3	cf_fir_12_16_10	12	13	935	935	935	12th order FIR filter
	4	cf_fir_7_16_16	7	8	619	619	619	7th order FIR filter
	5	cf_fir_3_8_8	3	4	148	148	148	3rd order FIR filter
	5bis	fm_transmitter	18	0	2671	1503	1182	FM transmitter

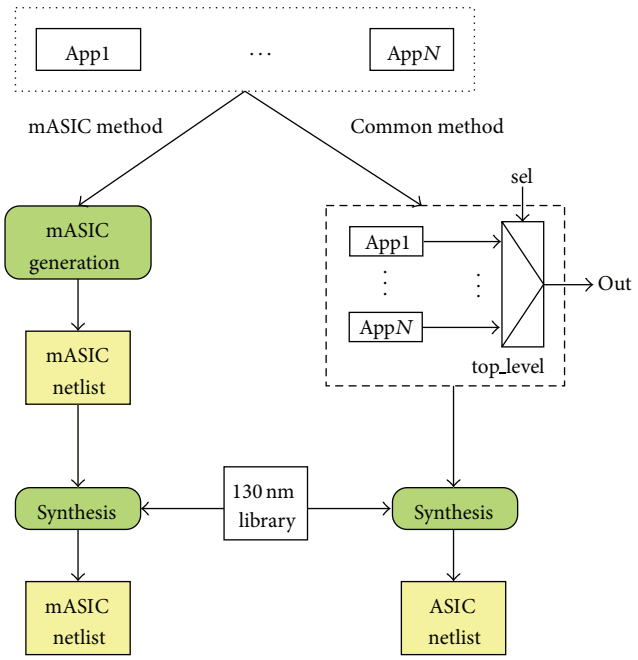


Figure 16: Synthesis methods used in this work.

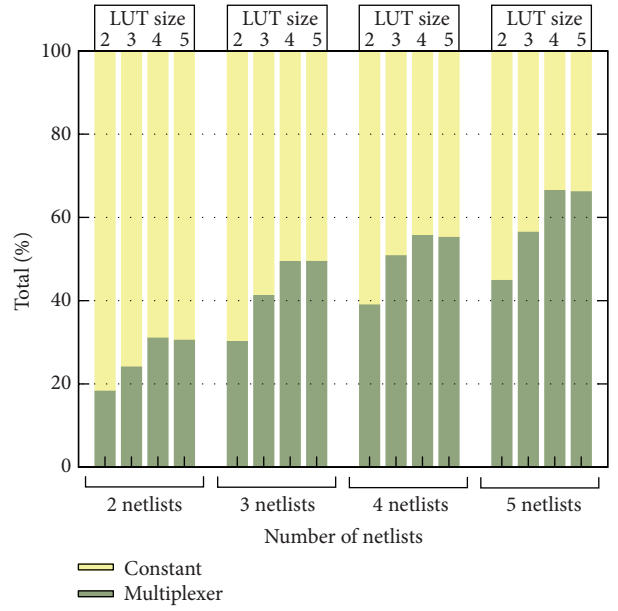


Figure 17: Percentage of replaced SRAM distribution for mASICs.

10–20% area [22]. However, in an ASIF, the occupation of the logic area increases to 40% because unused routing resources are removed. That is why it is better to use 2 input LUTs in ASIF.

In order to explore the effect of LUT size K (number of LUT inputs) on mASICs, the same experiments are done using LUT-2, LUT-3, LUT-4, and LUT-5 versions of MCNC netlists. We create randomly different combinations of 2, 3, 4, and 5 netlists generated using the mASIC generator. Then, we take the average results of mASICs which have the equal number of netlists and LUT size to evaluate the effect of the LUT size. These techniques allow us to justify our results with different netlists.

It should be remembered that even though look-up tables are used at the beginning of the mASIC generation flow, all SRAMs are replaced in the customization process by hard-coded bitstreams. They can be replaced either by a constant or a multiplexer which takes constants as inputs. Conditions are explained in Section 6. Obviously it is more advantageous if they are replaced by constants. The more mASIC has constants, the more constant propagation induces logic pruning and optimizes the area.

Figure 17 shows percentage of replaced SRAM distribution for mASICs with different number of netlists and different LUT sizes. There are two conclusions that we can draw from this figure. The first one is obvious: for each LUT size, as the number of netlists increases, SRAMs are replaced more with multiplexers instead of constants, because the probability of having same bit for all netlists decreases. The second one is the LUT size effect; it can be seen that a LUT-2 based mASIC has the highest percentage of constant among other LUT types. In a LUT-2 based mASIC-2, customization process replaces 73.5% of SRAMs by constants and the rest by multiplexers. In a LUT-5 based mASIC-2, the constant ratio

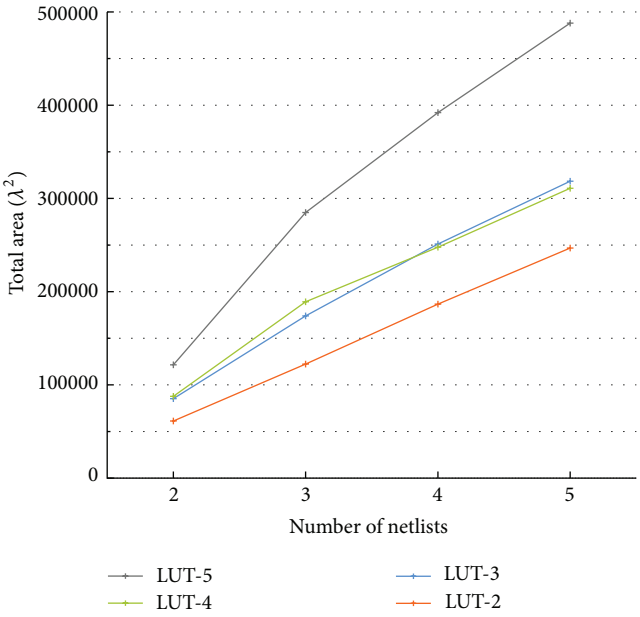


Figure 18: Total mASIC area comparison for different LUT size.

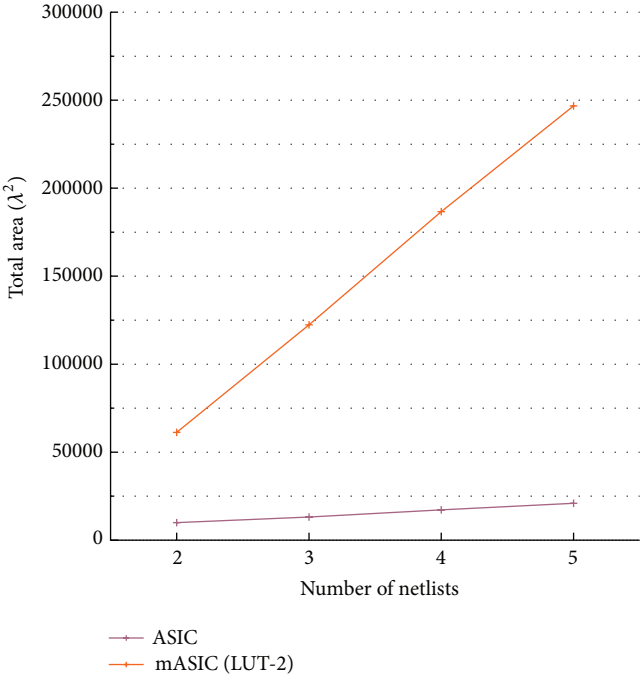


Figure 19: Total area comparison between ASIC and LUT-2 based mASIC.

decreases to 65%. It seems the LUT-2 based mASICs are more suitable for logic pruning.

To find out and compare the total area of mASICs based on different LUT sizes, their VHDL models are synthesized using Cadence RTL Compiler [18] and a 130 nm standard cell library. The graph in Figure 18 shows the total area of different mASICs in lambda square after synthesis. It turns out that the smallest mASIC area is obtained using LUT-2. This result is consistent with the conclusion that we have drawn from Figure 17. For 5 MCNC netlists, LUT-2 based mASIC is 2.5 times smaller than LUT-5 based mASIC. As we have noticed that the total area and the LUT input size are correlated, we ignored LUT size bigger than 5.

According to the experiments, the most efficient way of generating an mASIC is to use 2-input LUTs. However, an mASIC without macroblocks is far from being a better solution than the common synthesis method. Figure 19 shows the area comparison between LUT-2 based mASICs and ASICs for different numbers of netlists.

9.2. LUT Input Pin Reordering Effect. Previous results show the impact of the LUT size to total area. Also, they confirm that the more LUTs have higher percentage of constants, the more constant propagation induces logic pruning and optimizes the area. To increase the percentage of constants, a LUT input pin reordering technique is presented in Section 7. This technique serves to find more common bits in different netlist configurations placed on the same LUT by reordering its input pins. Later, common SRAM bits are replaced by constants. It should be remembered that this technique has 2 drawbacks.

(i) It can increase the routing area by increasing the number of multiplexers in the routing channel.

(ii) It is a brute force technique. Thus, it has a high function cost: $(n!)^N$ for each LUT on the initial architecture, where n is the number of LUT inputs and N is the number of netlists mapped on the LUT.

In this section we explore the impact of the LUT input pin reordering technique on the total area.

MCNC benchmarks shown in Table 1 are used in the experiments of this technique. We had already implemented mASICs with different LUT size and analyzed the constant-multiplexer ratio in CLBs in Figure 17. Here, we apply the LUT input pin reordering technique to mASICs which contain 2, 3, 4, and 5 netlists (mASIC-2, mASIC-3, mASIC-4, and mASIC-5). However, due to high function cost we could not retrieve the results for LUT-5 based mASIC-4 and mASIC-5 in a reasonable time.

An SRAM is replaced by a constant when all bitstreams of different netlists, which are using this particular SRAM, program it with the same value. This technique tries to increase the similarities between bitstreams of different netlists by altering LUT configurations. A LUT can have $n!$ different configurations where n is the number of LUT inputs. When the number of configurations increases, the possibility of finding similar bitstreams also increases. That is why the gain in terms of constants is correlated to LUT size. As the number of LUT inputs increases, the gain also increases. Figure 20 shows the percentage of constant increase in terms of LUTs after reordering. While in a LUT-2 based mASIC-3 there is only 1% more constants, in a LUT-5 based mASIC-3 this percentage reaches to 10%. However, this gain is not enough for LUT-5 to be a better solution than LUT-2. Figure 21 shows percentage of replaced SRAM distribution after reordering.

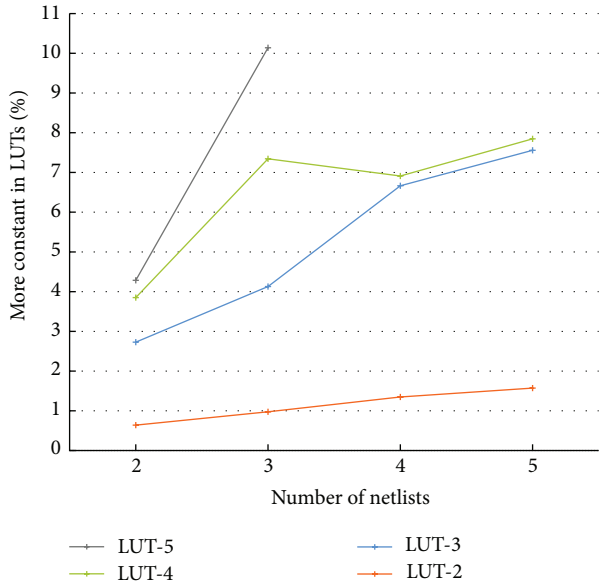


Figure 20: Constant increase in CLBs after reordering.

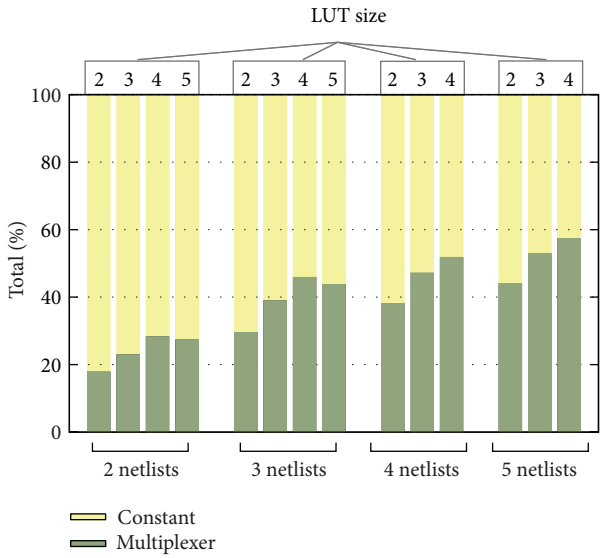


Figure 21: Percentage of replaced SRAM distribution after reordering.

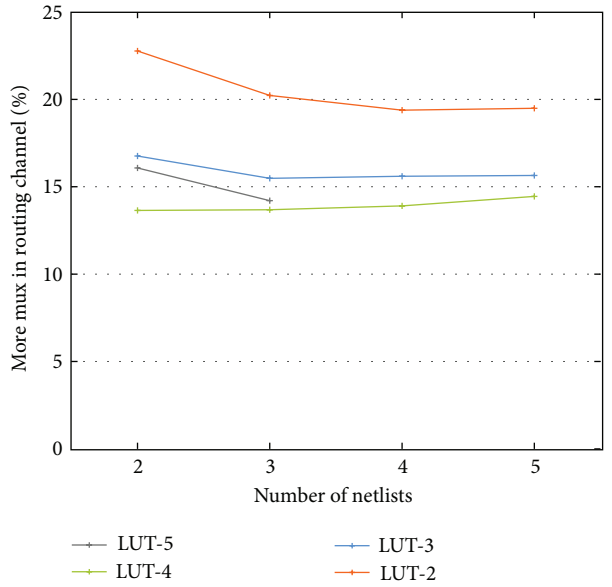


Figure 22: Multiplexer increase in routing channel after reordering.

It can be seen that even with the smallest gain from this technique, LUT-2 still has the highest percentage of constants.

The first drawback of this technique (function cost) prevented to get results from LUT-5 based mASIC-4 and mASIC-5. The second drawback is the increased routing area. If LUT input pin reordering technique is used, the router cannot modify the order of input pins in order to optimize the number of multiplexers. Thus, the number of multiplexers used by the router increases. The increase rate depends on the LUT size. In Figure 22 the multiplexers increase the routing channel in percentage for different LUT sizes. Based on the experiments, this percentage increases when the LUT size gets smaller. It is related to the fact that, in FPGAs and in ASIFs, when the LUT size decreases, the routing area increases [7]. It is also the case in mASIC. A LUT-2 based mASIC contains more CLB instances than an equivalent LUT-5 based mASIC. Hence, it needs more wires and multiplexers to route these instances. When there are more routing resources, the increase becomes more important. The worst case is the LUT-2 based mASIC-2. There are 22% more multiplexers. The best case is the LUT-4 based mASIC-2: The number of multiplexers in the routing channel increases 13.6%.

To evaluate the changes in LUTs and in the routing channel, we synthesized the generated VHDL model of optimized mASICs based on different LUT types using the same setup as the previous section to compare their area. The graph in Figure 23 shows the comparison of total areas of before and after reordering techniques of mASICs. The LUT optimization method is attractive when the LUT input size is bigger than 3. A LUT-5 based mASIC-2 gets 30% smaller after the optimization in terms of total area. However bigger LUT sizes with a number of netlists superior to 3 have a huge function cost. That is why the total area of LUT-5 versions of mASIC-4 and mASIC-5 could not be retrieved. The input pin reordering technique has a tiny impact on small-sized LUT based mASICs. It may increase or decrease the total area. For example, a LUT-2 based mASIC-2 gets 3.8% smaller but a LUT-2 based mASIC-3 gets 4.8% larger. It can be seen in Figure 23 that the LUT size effect has more impact on total area than the LUT input pin reordering. As a consequence, in overall, nonreordered LUT-2 based mASICs remain the best solution in terms of area.

9.3. mASIC Using Heterogeneous Architecture. mASIC optimization methodology allows to share resources between mutually exclusive applications. Larger resources invoke more area reductions when they are shared. For example,

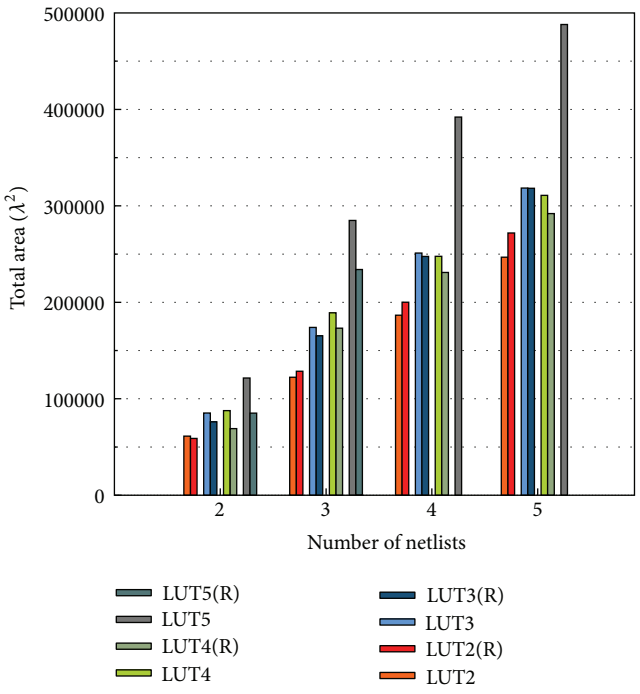


Figure 23: Total area comparison between before and after reordering.

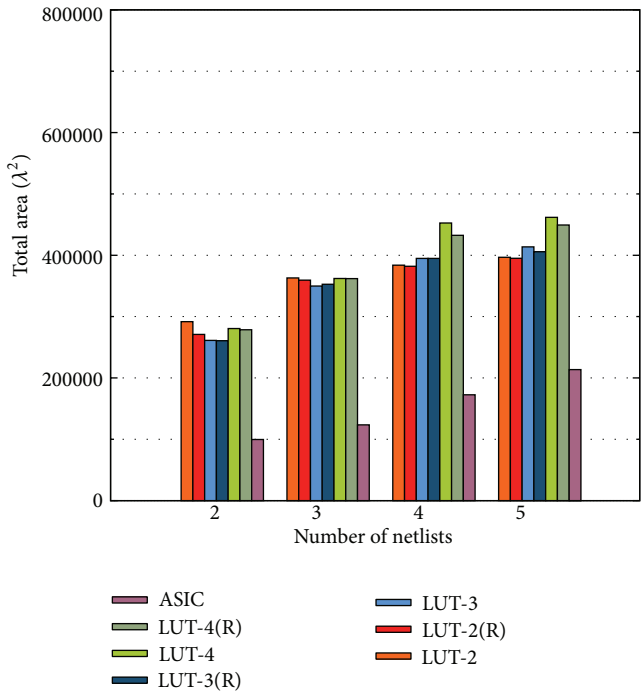


Figure 24: Total area comparison for OpenCores benchmarks [20] SET1.

it is more beneficial to share an 32-bit Adder rather than a CLB which contains a 4-input LUT and a flip-flop. Previous experiments show that the common synthesis method is more efficient in terms of area than the mASIC optimization methodology when fine-grained homogeneous architecture is (CLB based) used as an initial architecture. In this section, we introduce 2 types of macroblocks to the initial architecture: adder and multiplier. The experiments show that macroblocks have a large influence on the efficiency of the mASIC optimization methodology.

Table 2 shows 2 sets of OpenCores [20] benchmarks which are used to evaluate mASIC. The first set (SET1) regroups 5 different configurations of a single application and 1 different application. The second set (SET2) consists in combining different applications. As we have found out that smaller LUT sizes are more interesting for mASIC, we ignored LUT-5 and netlists are generated with LUT-2, LUT-3, and LUT-4. Multipliers and adders are tagged as hard blocks. Details regarding the conversion of these benchmarks (netlists) from HDL format to *.net* format are already described in Section 4.

For both benchmark sets, we compare total areas provided by different optimization methods and present the results in Figures 24 and 25. Different optimization methods are presented below:

(i) LUT-2, LUT-3, and LUT-4 based mASICs: *LUT-N*.

(ii) LUT-2, LUT-3, and LUT-4 based mASICs using LUT input pin reordering presented in Section 7: *LUT-N(R)*.

(iii) Common synthesis method using Cadence RTL Compiler [18]: *ASIC*.

The x-axis represents the number of netlists used in the experiment. For SET1, "2" means that *diffeq_c_systemC* and *cf_fir_16_16_16* are used, "3" means that *diffeq_c_systemC*, *cf_fir_16_16_16*, and *fm_receiver* are used, and so on. The same logic is also used for SET2 except for 5bis. "5bis" means that as the 5th netlist, instead of using *cf_fir_3_8_8* we use a different application: *fm_transmitter*. The order of netlists in Table 2 is met. The Y-axis represents the area in symbolic units (lambda square).

Figure 24 shows the total area comparison using SET1. In SET1, applications are different from each other and they contain considerable amount of soft blocks. This creates 2 problems. The first problem is the routing time. The more there are blocks to route, the more the top-down routing algorithm needs a larger routing channel width. With increasing number of blocks and increasing channel width, it may become impossible for the router to finish routing in a reasonable time. The LUT input pin reordering is also performed. It turns out that the common ASIC synthesis method gives the best results. A LUT-2 based mASIC for 5 netlists is 1.8 times larger than an ASIC. Even hard blocks are shared successfully and help to reduce the area, customization, and the constant propagation stage cannot manage to provide an efficient logic pruning for soft blocks. It is because there are huge amounts of soft blocks and their functions are different from each other.

In SET2 we have chosen netlists which are different configurations of the same application. Figure 25 shows the total area comparison using SET2. In SET2, like hard blocks,

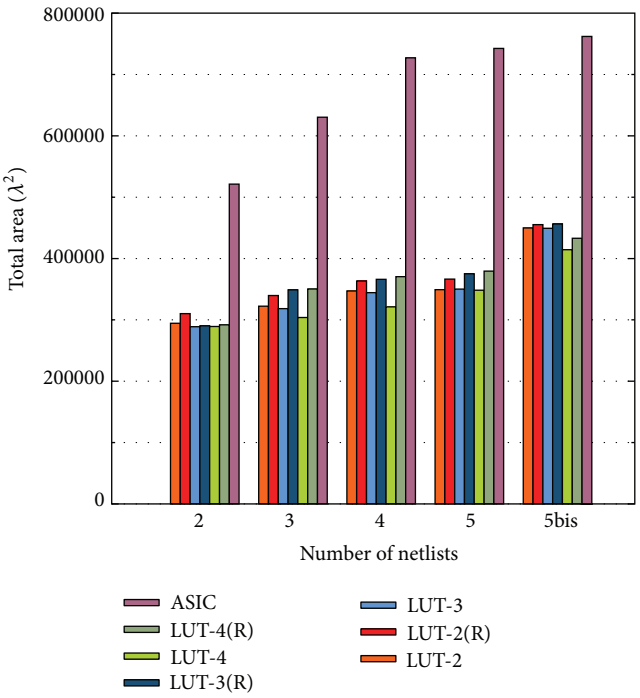


Figure 25: Total area comparison for OpenCores benchmarks [20] SET2.

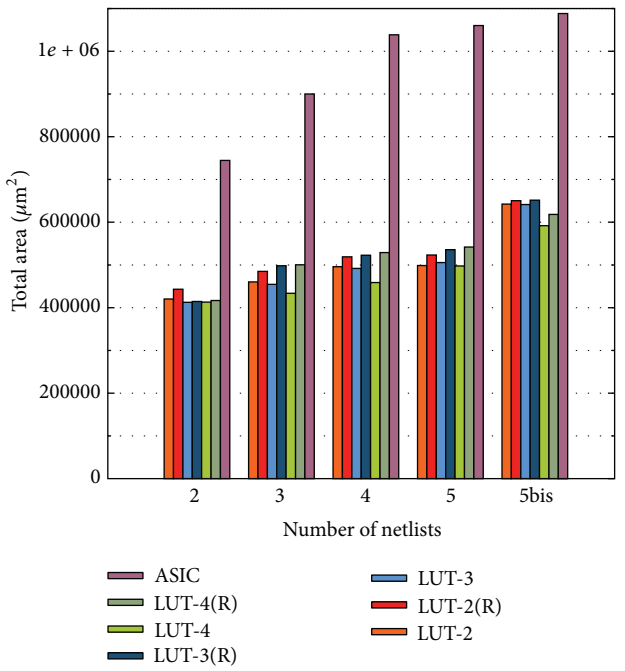


Figure 26: Total area comparison for OpenCores benchmarks [20] SET2 after place and route.

soft blocks are very similar to each other and they contain only 1 type of logic gate and flip-flops. There are several consequences of this fact. First, as there are no complex functions, there is no such difference between different LUT size in mASICs. Second, this creates an ideal situation for customization and constant propagation. Functions of different applications which are mapped in the same LUT will have more likely the same bitstreams. This increases the usage of constants to replace SRAMs in the customization process. As stated before, the more mASIC has constants, the more constant propagation induces logic pruning and optimizes the area. Third, as different bitstreams of a LUT are equal (or almost equal), LUT input pin reordering technique cannot find a better solution. On the contrary, it will increase the number of multiplexers on the routing network. Our experiments show that, by sharing hard blocks and soft blocks, mASIC optimization methodology generates 58% smaller circuit than the ASIC. Using a completely different netlist as the 5th netlist increases significantly the area (5bis in X-axis) but it remains smaller than ASIC. Until now, we have presented areas for a standard cell library after synthesis. It does not include the wire cost which is added after place and route. This is why we used an automatic place and route process with Cadence SoC Encounter [18] on SET2 where mASIC gives better results than a common synthesis method. The results are shown in Figure 26. It seems that, for SET2, wire cost remains insignificant and the area ratio between mASIC and ASIC does not change. However, for larger benchmarks, we expect that wire cost may become more important and decrease the area advantage of multi-mode systems.

As a consequence, we can see that using hard blocks allows mASIC generation to obtain smaller circuits. At the end, mASIC optimization methodology is an efficient method for similar netlists with a lot of hard blocks to share. When used for dissimilar netlists, results may become worse than the common ASIC synthesis method.

10. Conclusion

This paper presented an mASIC optimization methodology using efficient placement and routing algorithms. In this methodology, after the placement of input netlists on a predefined architecture with resource sharing, a joint netlist is created by routing logic blocks using available routing resources. Then, unused logic resources are removed from the placed and routed joint netlist. Later, all SRAMs are replaced by hard-coded bitstreams which allow logic pruning in the constant propagation stage. We also proposed a technique which increases similarities between bitstreams which are going to be hard-coded on the same LUT, to improve the efficiency of the constant propagation. Knowing that this technique has a negative impact on the number of multiplexers in the routing channel, we analyzed its effect on the total area.

Experiments show that the LUT size is correlated to the total area. For CLB based homogeneous netlists, LUT-2 gives the best results in terms of area. For 5 MCNC netlists (mASIC-5), LUT-2 provides 50% smaller circuit than LUT-5. It has been shown that reordering LUT inputs is more efficient with bigger LUT sizes but it has a limited usage due to very long execution time. Also the reordering technique increases the routing area significantly. That is why,

in overall, a nonreordered LUT-2 remains the best solution. However, without hard blocks, the circuit generated using the mASIC optimization methodology remains larger than the circuit generated using a common synthesis tool. When the experiments are performed on similar netlists which contains hard blocks such as multipliers and adders, it turns out that mASIC methodology can generate a circuit which is 53% smaller than an ASIC. This reveals that our method is efficient for similar applications.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgment

This work is partially funded by the ANR project ASTECAS.

References

[1] V. V. Kumar and J. Lach, "Highly flexible multimode digital signal processing systems using adaptable components and controllers," *EURASIP Journal on Applied Signal Processing*, vol. 2006, Article ID 79595, 9 pages, 2006.

[2] L.-Y. Chiou, S. Bhunia, and K. Roy, "Synthesis of application-specific highly efficient multi-mode cores for embedded systems," *ACM Transactions on Embedded Computing Systems*, vol. 4, no. 1, pp. 168–188, 2005.

[3] C.-Y. Huang, Y.-S. Chen, Y.-L. Lin, and Y.-C. Hsu, "Data path allocation based on bipartite weighted matching," in *Proceedings of the 27th ACM/IEEE Design Automation Conference*, pp. 499–504, ACM, June 1990.

[4] C. Andriamisaina, P. Coussy, E. Casseau, and C. Chavet, "High-level synthesis for designing multimode architectures," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 29, no. 11, pp. 1736–1749, 2010.

[5] E. Casseau and B. Le Gal, "Design of multi-mode application-specific cores based on high-level synthesis," *Integration, the VLSI Journal*, vol. 45, no. 1, pp. 9–21, 2012.

[6] K. Compton and S. Hauck, "Automatic design of area-efficient configurable ASIC cores," *IEEE Transactions on Computers*, vol. 56, no. 5, pp. 662–672, 2007.

[7] H. Parvez, Z. Marrakchi, A. Kilic, and H. Mehrez, "Application-specific fpga using heterogeneous logic blocks," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 4, pp. 24:1–24:14, 2011.

[8] S. Trimberger, D. Carberry, A. Johnson, and J. Wong, "Time-multiplexed FPGA," in *Proceedings of the 5th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 22–28, IEEE Computer Society, Washington, DC, USA, April 1997.

[9] N. Miyamoto and T. Ohmi, "Temporal circuit partitioning for a 90nm CMOS multi-context FPGA and its delay measurement," in *Proceedings of the 15th Asia and South Pacific Design Automation Conference (ASP-DAC '10)*, pp. 373–374, IEEE Press, Piscataway, NJ, USA, January 2010.

[10] "Tabula," http://www.tabula.com/.

[11] J. Luu, I. Kuon, P. Jamieson et al., "Vpr 5.0: Fpga cad and architecture exploration tools with single-driver routing, heterogeneity and process scaling," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 4, pp. 32:1–32:23, 2011.

[12] G. Lemieux, E. Lee, M. Tom, and A. Yu, "Directional and single-driver wires in FPGA interconnect," in *Proceedings of the IEEE International Conference on Field-Programmable Technology (FPT '04)*, pp. 41–48, IEEE, December 2004.

[13] "Flexras," http://www.flexras.com/.

[14] "Berkeley logic interchange format (blif)," 1996.

[15] E. Sentovich, K. Singh, L. Lavagno et al., "Sis: a system for sequential circuit synthesis," Tech. Rep. UCB/ERL M92/41, EECS Department, University of California, Berkeley, Calif, USA, 1992.

[16] A. Marquardt, V. Betz, and J. Rose, "Using cluster-based logic blocks and timing-driven packing to improve FPGA speed and density," in *Proceedings of the ACM/SIGDA 7th International Symposium on Field Programmable Gate Arrays (FPGA '99)*, pp. 37–46, ACM, New York, NY, USA, February 1999.

[17] E. Ahmed and J. Rose, "The effect of lut and cluster size on deep-submicron fpga performance and density," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 12, no. 3, pp. 288–298, 2004.

[18] "Cadence," http://www.cadence.com/.

[19] S. Yang, *Logic Synthesis and Optimization Benchmarks User Guide: Version 3.0*, Microelectronics Center of North Carolina (MCNC), 1991.

[20] "Opencores," http://www.opencores.org/.

[21] H. Parvez and H. Mehrez, *Application-Specific Mesh-Based Heterogeneous FPGA Architectures*, Springer, Berlin, Germany, 2011.

[22] V. Betz, J. Rose, and A. Marquardt, Eds., *Architecture and CAD for Deep-Submicron FPGAs*, Kluwer Academic Publishers,, Norwell, Mass, USA, 1999.