Leading the Blind: Automated Transistor-Level Modeling for FPGA Architects

THÈSE N^{O} 7928 (2017)

PRÉSENTÉE LE 4 AOÛT 2017
À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS
LABORATOIRE D'ARCHITECTURE DES PROCESSEURS
PROGRAMME DOCTORAL EN INFORMATIQUE ET COMMUNICATIONS

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Grace ZGHEIB

acceptée sur proposition du jury:

Prof. A. Argyraki, présidente du jury
Prof. P. Ienne, directeur de thèse
Prof. V. Betz, rapporteur
Dr J. Greene, rapporteur
Prof. Y. Leblebici, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE
Suisse
2017

Sometimes it is the very people
who no one imagines anything of
who do the things no one can imagine.
— Christopher Morcom

Acknowledgments

We often need such big milestones in our lives to force us to stop and look back. It is not about searching for memories, as much as it is about assessing the journey with its every curve, crossroad, and mountain top; and with every traveler whose path crossed ours along that long winding road. Some we recognize as passersby that we once exchanged traveling stories with, while others still have their footsteps marked alongside ours, for as far back as we can see.

It goes without saying that the most influential presence throughout this journey was that of my supervisor, Prof. Paolo Ienne. I have always considered myself lucky to get the chance to not only work with Paolo, but also to learn from him. His invaluable input helped me to, as he called it on our first meeting, find my path through the foggy world of research.

I also got the chance to work alongside many great minds, such as Dr. Hadi Parandeh-Afshar who was the first to show me what lies under the hood of FPGAs, Dr. David Novo whose insight and help were truly vital, as well as Dr. Madhura Purnaprajna and Dr. Muhsen Owaida.

But of course, it is more than just about work. It is also about the friends I made along the way, especially the current and former LAP members, namely Ali, Ana, André, Andrew, Chantal, Lana, Giulia, Mikhail, Nithin, René, Robert, Sahand, Stefan, and Thomas. With special mentions to Dr. Ana Petkovska, with whom I shared not only an always-flourishing office but also special moments of both struggle and success, in the past 6 years; to Lana Josipovic, whose spontaneous hugs and our long hours of chatting about anything and everything will certainly be missed; and, to LAP's super-secretary Chantal Schneeberger, who became my go-to person, capable of always finding the most convenient solutions.

I am blessed beyond measures to have a lifelong friend, Dr. David Kozhaya, with whom I walked the last thirteen years of my life. Having such a strong reliable support is a gift to which I will always be grateful. Even if our paths get separated, I know that he will always be there for me. Special acknowledgments go to those who stood by me through the years, sometimes even through long distances, especially Ghassan Stefan, Mirella Abboud and Dr. Serj Haddad.

Finally, there could be no stronger support system than that of the family: my mother and father, my sister Josiane, and Ziad, who keep on offering, every step of the way, their unconditional love and crucial support.

You have all left your marks, some deeper than others, each in his way, and for that I am grateful to each and every one of you!

Lausanne, July 7th, 2017 G. Z.

Abstract

The design and development of innovative FPGA architectures hinge on the flexibility of its toolchain. Retargetable toolchains, like the Verilog-to-Routing (VTR) flow, have been developed to enable the testing of new FPGAs by mapping circuits onto easily-described and possibly theoretical architectures. However, in reality, the difficulty extends beyond having CAD tools that support the architectural changes: it is equally important for FPGA architects to be able to produce reliable delay and area models for these tools. In addition to having acute architectural intuitions, designing and optimizing the circuit at the transistor-level requires architects to have, as well, a particular set of electrical engineering skills and expertise. The process is also painstaking and time-consuming, rendering the comparison of a variety of architectures or the exploration of a wide design space quite complicated and even impossible in practice.

In this work, we present a novel approach to model the delay and area of FPGA architectures with various structures and characteristics, quickly and with acceptable accuracy. Abstracting from the user the transistor-level design and optimization that normally accompany the modeling process, this approach, called FPRESSO, can be used by any architect without prerequisites. We take inspiration from the way a standard-cell flow performs large-scale transistor-size optimization and apply the same concepts to FPGAs, only at a coarser granularity. Skilled designers prepare for FPRESSO a set of locally optimized libraries of basic parameterizable components with a variety of drive strengths. Then, inexperienced users specify arbitrary FPGA architectures as interconnects of these basic components. The architecture is globally optimized, within minutes, through a standard logic synthesis tool, by choosing the most fitting version of each cell and adding buffers wherever appropriate. The resulting delay and area characteristics are automatically returned, in a format suitable for the VTR flow.

A correct modeling of any architecture requires not only an optimization of the logic components, but also a proper modeling of the wires connecting these components. This does not only include measuring the length of the wires to determine their respective resistance and capacitance, but also, minimizing their length to reduce the wireload effect on the overall performance. To that end, FPRESSO features an automatic and generic wire modeling approach based on a simulated annealing floorplanning algorithm, to estimate the wires between the different components of the FPGA architecture.

To evaluate the results of FPRESSO and confirm the validity of its modeled architectures, we use it to explore a wide range of FPGA architectures. First, we repeat a known study that helped set the standards on the optimal Look-Up-Table (LUT) and cluster size for conventional FPGAs.

Abstract

We show, by comparing with the results of the study, that modeling in FPRESSO preserves the very same trends and conclusions, with significantly less effort. We then extend the search space to cover fracturable LUTs and sparse crossbars, and show how FPRESSO makes the exploration of a huge search space not only possible but easy, efficient, and affordable, for any class of VTR users.

Key words: Field Programmable Gate Array, FPGA, architecture design, architecture modeling, transistor design, architectural exploration, FPRESSO, wireload modeling, CAD tools.

Résumé

La conception et le développement d'architectures de FPGA innovantes dépendent de la flexibilité de la chaîne d'outils. Des chaînes d'outils ciblées, comme le *Verilog-to-Routing (VTR)*, ont été développées pour permettre de tester de nouvelles FPGAs en implémentant des circuits sur des architectures faciles à décrire et potentiellement théoriques. La difficulté pour les architectes de FPGA n'est pas seulement d'avoir des outils CAD flexibles qui supportent les changements d'architecture, mais également d'être capables de produire des modèles de délai (*delay*) et de surface (*area*) pour ces outils. De bonnes intuitions architecturales ne sont cependant pas suffisantes, en effet la conception et l'optimisation des circuits au niveau des transistors nécessitent également un ensemble de compétences et spécialisations d'ingénierie électrique particulières. Le processus est méticuleux et long, ce qui rend la comparaison d'une grande variété d'architectures et l'exploration d'un large espace de conception plutôt compliqué et même impossible en pratique. Dans ce travail, nous présentons une nouvelle approche pour modeler, rapidement et avec une précision acceptable, le délai et la surface des architectures de FPGA ayant des structures et caractéristiques diverses. En masquant à l'utilisateur la conception au niveau des transistors et l'optimisation qui accompagnent normalement le processus de modélisation, cette approche appelée FPRESSO peut être utilisée par tout architecte sans conditions préalables. Nous nous inspirons de la façon dont un flux de cellules standard (*standard-cell flow*) effectue des optimisations à grande échelle de tailles des transistors et nous appliquons les mêmes concepts aux FPGAs, mais avec une plus grande granularité. Des designers qualifiés préparent et optimisent localement des librairies de composants de base paramétrables ayant une variété de *drive strengths*. Puis, des utilisateurs inexpérimentés spécifient des architectures arbitraires de FPGA en utilisant ces composants de base interconnectés. L'architecture est globalement optimisée, en quelques minutes, par un outil de synthèse logique standard qui choisit la version la plus appropriée de chaque composant et ajoute des *buffers* si nécessaire. Les caractéristiques de délai et surface qui en résultent sont automatiquement retournées dans un format approprié pour le flux VTR.

Une modélisation correcte de toute architecture exige, non seulement une optimisation des composants logiques, mais aussi une modélisation appropriée des fils qui connectent ces composants. Cela ne comprend pas seulement la mesure de la longueur des fils pour déterminer leur résistance et capacité relatives, mais aussi la minimisation de cette longueur pour réduire l'effet de la capacité des fils (*wireload effect*) sur la performance globale. À cet effet, FPRESSO dispose d'une approche automatique et générique de modélisation de fils (*wire modeling*) basée sur un *simulated annealing* algorithme de *floorplanning*, pour modéliser

correctement les fils connectant les différents composants de l'architecture de FPGA.
Afin d'évaluer les résultats d'FPRESSO et de confirmer la validité de ses architectures modélisées, on l'utilise pour explorer une large gamme d'architectures de FPGA. Premièrement, nous reproduisons une étude bien connue qui a aidé à établir les normes existantes de la taille optimale du *Look-Up-Table (LUT)* et du *cluster* des FPGAs conventionnelles. Nous démontrons, en comparant les résultats d'FPRESSO avec ceux de l'étude, que notre modélisation conserve les mêmes tendances et conclusions, et ceci beaucoup plus rapidement et avec beaucoup moins d'effort. Nous augmentons ensuite l'espace de recherche pour inclure des *fracturable LUTs* et des *sparse crossbars*, et démontrons comment FPRESSO rend l'exploration d'un vaste espace de recherche non seulement possible mais également facile, efficace et abordable, pour toute classe d'utilisateurs de VTR.

Mots clefs : Field Programmable Gate Array, FPGA, conception d'architecture, modélisation d'architecture, conception de transistors, exploration architecturale, FPRESSO, modélisation de la capacité des fils, outils de CAD.

Contents

List of Figures

List of Tables

1 The Challenges of FPGA Architectural Exploration

The *Field Programmable Gate Arrays (FPGAs)* market has been expanding fast, bringing the many benefits of reconfigurable computing to new domains such as cloud and mobile computing. With this increased interest in FPGAs comes a stronger demand to further bridge the existing efficiency gap between FPGAs and dedicated circuits, coupled with strong energy-efficiency requirements.

Intuitively, these demands need to be addressed at the hardware level, by improving the architecture of the FPGAs. The improvements can come from minor modifications to the existing logic and routing, or through completely new and unconventional architectures. Given the intrinsic difficulty in having a single FPGA architecture that realistically satisfies most requirements, it is common to have a spectrum of architectures, each targeting a particular application or niche.

A software flow able to synthesize circuits onto a wide range of different FPGA architectures is clearly essential in enabling proper architectural explorations. Fortunately, the *Verilog-to-Routing (VTR)* project [Rose et al., 2012] already provides such a retargetable flow supporting a wide variety of easily-described hypothetical FPGA architectures. Since its introduction in 2012, hundreds of research projects have relied on VTR to evaluate new applications or architectural modifications. More so, precursors of VTR have been successfully used in industrially plausible FPGA architectures to explore optimal logic block sizes [Ahmed and Rose, 2004] and to show area and delay tradeoffs [Kuon and Rose, 2011]. Its latest official release, VTR 7.0 [Luu et al., 2014a], added hard adders and carry chain support, enabling a detailed exploration of efficient architectures for applications with heavy arithmetic operations [Luu et al., 2014b]. Furthermore, VTR has also been used to explore radically new architectures such as those based on *And-Inverter Cones (AICs)* [Parandeh-Afshar et al., 2012, 2013; Zgheib et al., 2014]. In such cases where a new logic block is introduced, the earlier stages of the tool flow, typically the technology mapper, might require modifications to support the new logic. However, the flow is modular enough to easily incorporate new algorithms like the depth-constrained technology mapper for AICs [Jiang et al., 2015]. The later stages of the flow, like the packer, are generic and can deal with any hypothetical architecture, as long as it can

be described in what is known as an architecture file.

The caveat, however, is that VTR architecture files need reasonably accurate area and delay models if the results are to be meaningful and the comparison between architectures reliable.

1.1 Modeling Challenges

Creating a good model of a hypothetical FPGA architecture is challenging, due to the difficulty in predicting the effect of transistor-level optimizations on the circuit. Two elements change dramatically the area and delay characteristics of an architecture: appropriate transistor sizing and correct signal buffering. Even if the design of the logic itself remains generally the same in the FPGA, both these elements critically depend on the used technology node and on the architecture being explored. They are directly affected by the architectural changes, no matter how minor they might seem, such as the number or size of the *Look-Up Tables (LUTs)*, the number of cluster inputs or crossbar switches, etc.

Implementing the required transistor-level optimizations to obtain reliable estimates of area and delay is a significant challenge due to the sizes of the circuits at hand and the particular set of electrical engineering expertise it requires. This has a direct impact on the feasibility of wide search-space architectural explorations. For instance, the latest studies on the optimal architecture parameters (mainly LUT and cluster size) for delay and area [Ahmed and Rose, 2000, 2004], limited their explorations to about 60 architectures due to time and feasibility constraints. Manually sizing the design and running SPICE simulations for every modeled circuit is a time-consuming process that is highly restrictive for any architectural exploration, not to mention the consistency and reproducibility issues it can cause.

However, one can argue that, for standard architectures that have already been studied extensively [Kaptanoglu et al., 1999; Rose et al., 1990; Singh et al., 1992], the designer might have already a good starting point through existing references that help narrow the search space and probably predict the outcome of minor changes in this kind of standard architectures. The problem becomes more critical in the cases where a totally new and unconventional logic block is introduced. Designing a completely revamped architecture around this new logic can be a major challenge. Designers do not have any guidelines that can help narrow the search for the optimal parameters: logic block and cluster size, routing density and connectivity, etc. We experienced this, first hand, when attempting to improve the design of the And-Inverter Cones, for example, and realized the difficulties and complexity coupled with the search for the optimal architecture that suits these logic blocks [Zgheib et al., 2014]. An experienced engineer, specialized in transistor-level circuits, designed and manually optimized each architecture, using SPICE simulations to evaluate it. This rendered the search highly inefficient and limited it to a handful of architectures.

There is no denying that manual sizing and optimization of the transistor-level circuits can lead to well-targeted and potentially optimal results, far better than any automatic modeling

approach. Nevertheless, the lengthy process of manual sizing and simulations, as well as the particular skill-set it requires—and that not all FPGA architects have—makes it impractical for a fast evaluation of new architectures. Thus, with this overwhelming evidence of the difficulty architects face in evaluating new architectures, and its direct impact on potential architectural explorations, it has become increasingly critical to find an easy and fast way to model FPGAs. The objectives are clear: to allow architects who do not necessarily have any transistor-level expertise to quickly model the delay and area of any hypothetical or wildly unconventional FPGA architecture, with reasonable accuracy.

1.2 Existing solutions and their limitations

Evidently, these modeling challenges are not new and researchers have already tried to address the problem through different attempts and approaches.

An intuitive approach would be to use a semicustom design flow, based on standard cells, to design the FPGA from a register-transfer level description. The results of such a semicustom flow can be used as conservative estimates of what good designers could conceive at transistor-level. Actually, designers have discovered over the years that for many complex components (e.g., fast arithmetic components) semicustom approaches are today even superior to hand-crafted circuits [Eriksson et al., 2003]—and thus represent perfect estimates of what is achievable. Unfortunately, for FPGAs, Kim and Anderson [2015] have recently shown that FPGA architectures designed with standard cells still incur severe area and delay overheads when compared to commercial full-custom FPGAs. Furthermore, these overheads have large variations across FPGA components rendering the models hardly faithful and thus unusable to drive realistic FPGA architecture explorations.

There have been several attempts to automate the transistor sizing problem for custom circuits by formulating it as an optimization problem with a specific objective function which generally tends to be the minimization of the delay and/or area. It goes back to more than 30 years ago when TILOS [Fishburn and Dunlop, 1985] was introduced as an automatic transistor sizing tool for custom circuits. TILOS' heuristic iteratively identifies the critical path of the circuit and increases the sizes of the transistors along that path until the optimization objectives are met. The authors show that the transistor sizing problem can be transformed into a convex optimization problem by modeling the transistors as a linear set of resistors and capacitances, while using the Elmore [Elmore, 1948] and Penfield-Rubinstein [Rubinstein et al., 1983] models to calculate the delay. However, despite its convexity, the resulting optimization can end up being suboptimal, while the used linear models and the Elmore delay have been shown to be inaccurate [Kasamsetty et al., 2000; Ousterhout, 1984]. Thus, to increase the modeling accuracy, new algorithms tried to use time-domain simulations to estimate the delay, but at a high computational complexity cost [Conn et al., 1998, 1999]. However, there are fundamental differences between the transistor-level optimization of custom non-configurable integrated circuits and reconfigurable circuits, like FPGAs, which present unique optimization challenges.

Kuon and Rose [2011] address these challenges in a two-phased algorithm that still uses linear models (similar to TILOS) in the first phase through an exploratory transistor-sizing heuristic, but then the second phase corrects the inaccuracy of these linear models by fine tuning the previously sized transistors using SPICE simulations. Nevertheless, the process remains time consuming, relies mainly on the inaccurate linear models, and is not publicly available for the research community.

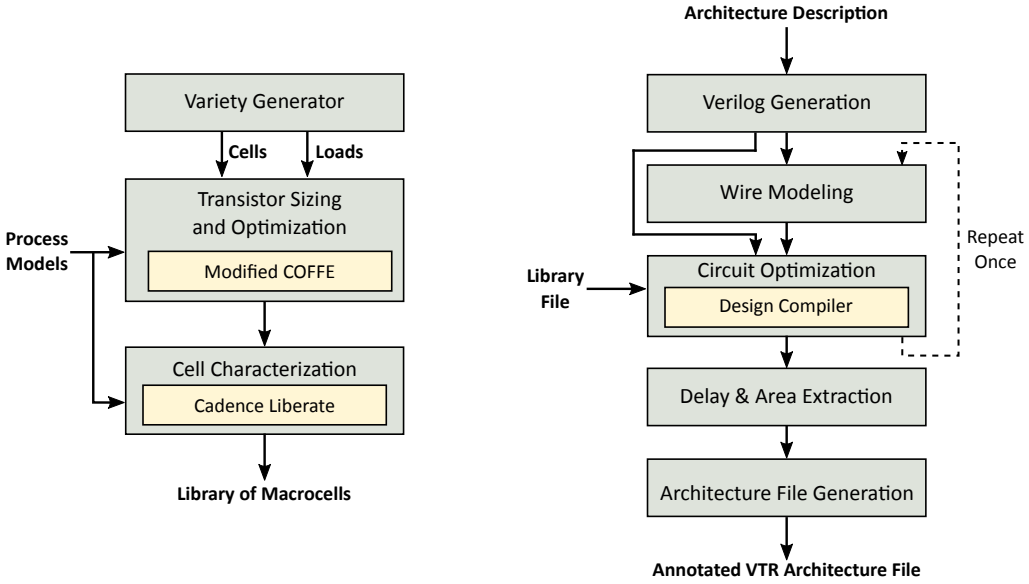
The most recent automatic transistor sizing tool specific for FPGAs, COFFE [Chiasson and Betz, 2013], addresses the limitations of its predecessors by phasing out the linear models and relying entirely on SPICE simulations, while improving the area and wireload models. It focuses on reducing the complexity of the transistor-size optimization in an FPGA by exploiting the structure of the circuit and by implementing ad hoc but efficient optimization strategies. The result is a fixed, yet parameterizable, architecture built at transistor level and a set of scripts implementing programmatically the required optimization for specific parameters, using appropriate SPICE simulations for measurement. Although this works quite well for the given standard parametric architecture supported by COFFE, the optimization process is quite slow (in the order of hours). More importantly, there is no support for other quite different architectures researchers might want to experiment with: the optimization strategy is built into the scripts that constitute COFFE and, although in principle adaptable, porting it to wildly different architectures might essentially mean rewriting the tool from scratch, albeit with an excellent starting point. It would also require a certain level of transistor design and optimization expertise that the FPGA architects do not necessarily have.

1.3 Our Approach

In this work, we address the FPGA modeling problem by proposing a novel approach that facilitates correct and quick modeling of complete FPGA architectures for users who are not necessarily transistor-level circuit designers. Our approach, called FPRESSO[1], is able to model with an acceptable accuracy the delay and area of a wide range of largely different FPGA architectures without requiring the users to understand the issues of transistor sizing. For most users, all that is required is a topological description of the cluster of an FPGA and the automatic results are VTR architecture files annotated with area and timing estimations.

Our approach to making sound optimizations of complete FPGA architectures is somehow modeled on the well-known divide and conquer approach used in semicustom design: Firstly, transistor-level designers construct highly-optimized libraries of standard building blocks (the standard cells). Libraries do not only limit the functionality of the cells to a set of basic classes, but contain several replicas of the same cell spanning a wide variety of transistor sizes. Secondly, once a library is available in a given technology, the cells are characterized, measuring in detail their area and delay characteristics. Finally, logic synthesizers, besides logically restructuring the target design and implementing it using the available standard cells,

[1] FPGA express modeling technique



(a) Library generation flow. **(b)** Main flow.

Figure 1.1 – The two main parts of FPRESSO's tool flow. (a) The library generation flow where experienced designers build, offline, a library of precharacterized components required in the main flow; (b) the main flow where inexperienced users provide an arbitrary description of the architecture and FPRESSO models its timing behavior using the library of components, before returning a VTR-compatible architecture file complete with all required area values and timing arcs.

choose the most appropriate yet functionally equivalent cells and add the required buffers to meet detailed high-level delay or area constraints. From the optimization point of view, the key of the semicustom design process is in locally optimizing transistors within the cells to display a variety of potentially useful timing behaviors (a process that is performed by transistor-level experts, once per technology node), and then in optimizing the overall circuit at a much higher abstraction than individual transistors and SPICE simulations (a process performed by every digital designer, for every designed circuit).

FPRESSO does exactly the same, conceptually, but avoids the unacceptable modeling errors of using standard cells by changing the granularity of the process. Instead of using standard cells to construct the circuit, FPRESSO relies on libraries of macrocells which consist of typical components that can be found in common FPGA architectures. Actually, our procedure resembles so much the classic standard-cell design flow that, in our experimental implementation, we strive to use wherever possible widely available off-the-shelf tools. Yet, the fact that our granularity is quite different (LUTs instead of NAND gates, so to speak) creates challenges in every step of our conceptually simple flow.

The flow can be split into two main, yet independent phases, as depicted in Figure 1.1. In the first phase, namely the library generation phase, expert designers construct at transistor-level all the components typically found in FPGA architectures, such as LUTs, crossbars, multiplexers, flip-flop assemblies, etc. An automated procedure generates a variety of implementations

in terms of transistor sizes (e.g., drive strengths) and optimizes all versions of the components. The state-of-the-art academic FPGA transistor-sizing tool, COFFE, is customized to automate this process and interface with the next stage of the library generation flow, as shown in Figure 1.1a. Then, in the cell characterization step, SPICE-like simulations extract the delay and area characteristics of the cells to build a library of macrocells. Both the optimization and characterization steps are slow and time-consuming procedures that require a level of expertise not every user has, but, as long as no new functional blocks are introduced, the library is built only once per technology node, offline and reused as many times as needed in the main flow of FPRESSO. Chapter 3 explains the details of the library generation phase with its cell optimization and characterization steps, as well as the challenges faced in adapting industry-standard methodologies for standard-cell characterization to operate correctly at the level of granularity of our macrocells.

The second phase of FPRESSO consists of the main flow that is actually exposed to the users. A completely automated process takes from inexperienced users a topological description of the FPGA architecture and optimizes it to generate reliable models of the achievable area and delay. The architecture is essentially a circuit, typically composed of a few hundreds of predefined cells, so the optimization simply selects from the library of cells (generated in the first phase) the ones with the appropriate drive strengths and adds buffers wherever needed. Once the architecture is optimized, static timing analysis is performed to extract all the relevant timing details. The result is a VTR-compatible architecture file, annotated with the respective delay and area measurements. It is important to realize that this second phase, shown in Figure 1.1b and detailed in Chapter 5, demands no more transistor-level knowledge than specifying the cluster topology for VTR and yet is not limited to particular, predefined architectures. The process is extremely fast (in the order of minutes) and entirely automated, completely abstracting any complexity from the user.

A correct architecture modeling does not only include the modeling of the logic and routing components that compose the FPGA architecture, but also the modeling of the wires that connect these components. To that end, FPRESSO uses a simulated-annealing floorplanning algorithm to minimize the total wire length. When the floorplan is optimized, it estimates the resistance and capacitance of each wire and adds it to the architecture optimization phase. Chapter 4 explains the different steps of the wire modeling and floorplanning algorithm.

Finally, to evaluate the accuracy of our modeling approach and showcase its efficiency in large-scale architectural explorations, we use FPRESSO to repeat and extend the latest study on the optimal FPGA architecture. The expanded search space includes sparse crossbars and fracturable LUTs. The results, depicted in Chapter 6, show that explorations where architectures are modeled either manually or with FPRESSO reach exactly the same conclusions, which validates the correctness of our modeling approach.

However, before elaborating more on the details of the different parts of FPRESSO, we first present a brief introduction, in Chapter 2, of the FPGA architecture and parameters, its CAD

flow, and the experimental setup used in all our architecture explorations.

2 Background Information

Before going over the details of our architecture modeling approach, we will first cover some fundamental characteristics of the FPGA architecture, its CAD flow, and the experimental setup used in our architecture explorations.

2.1 FPGA Architecture

State-of-the-art FPGAs adopt an island-style architecture that consists of a grid of logic clusters connected though vertical and horizontal routing channels [Betz et al., 1999; Kuon et al., 2008], as shown in Figure 2.1. In this section, we overview the main characteristics of the logic cluster and the parameters that define it. We also discuss two key features of the FPGA architecture, namely the fracturable LUTs and the depopulated crossbars, which will be used and explored in the subsequent chapters.

2.1.1 General Cluster Architecture

Our modeling approach does not target a specific FPGA architecture but is generic and can support any logic cluster, as long as it is composed of elements that exist in its library. However, over the next few chapters, we will base our experiments and explorations on an FPGA architecture with a structure similar to that of the Altera Stratix FPGAs [Lewis et al., 2005]. This is mainly due to the fact that the existing studies on optimal architecture parameters and the tools that we can compare to are based on this type of FPGA.

Figure 2.2 shows the general structure of the logic cluster (also referred to as *Complex Logic Block (CLB)*) along with the different parameters that can define a particular architecture. Each cluster consists of N *Basic Logic Elements (BLEs)* and has I inputs and N outputs. Each BLE has a K-input LUT, a register and two multiplexers to select between the registered and unregistered LUT output, before sending it either to the cluster output or as a local feedback. The I inputs, along with the N feedback signals, feed the input crossbar which then distributes them to the BLEs (and hence the LUTs). As such, the crossbar has $(I + N)$ inputs, $(N \times K)$

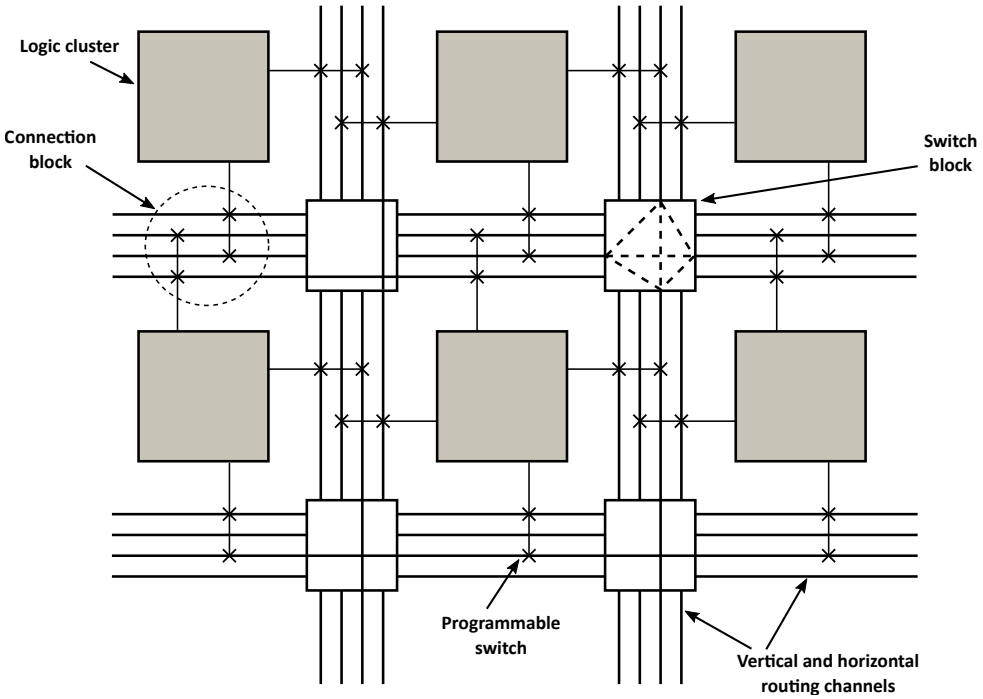


Figure 2.1 – The island-style FPGA architecture.

outputs, and a density $F_{c_{local}}$ which indicates the fraction of the inputs connected to each crossbar output.

The inputs and outputs of the cluster are connected to the global routing through Connection Blocks (CBs) and Switch Blocks (SBs), respectively. The fraction of routing channels connected to each of the cluster's inputs and outputs is defined by the parameters $F_{c_{in}}$ and $F_{c_{out}}$, respectively; while the total number of routing tracks on either side of the cluster is determined by W.

Table 2.1 summarizes all these parameters that can be used to define a specific architecture or determine the search space of architectural explorations.

Table 2.1 – The FPGA architecture parameters.

Parameter	Description
K	LUT size
N	Cluster size
I	Number of cluster inputs
$F_{c_{local}}$	Input crossbar density
$F_{c_{in}}$	Cluster input connection flexibility
$F_{c_{out}}$	Cluster output connection flexibility
W	Routing channel width

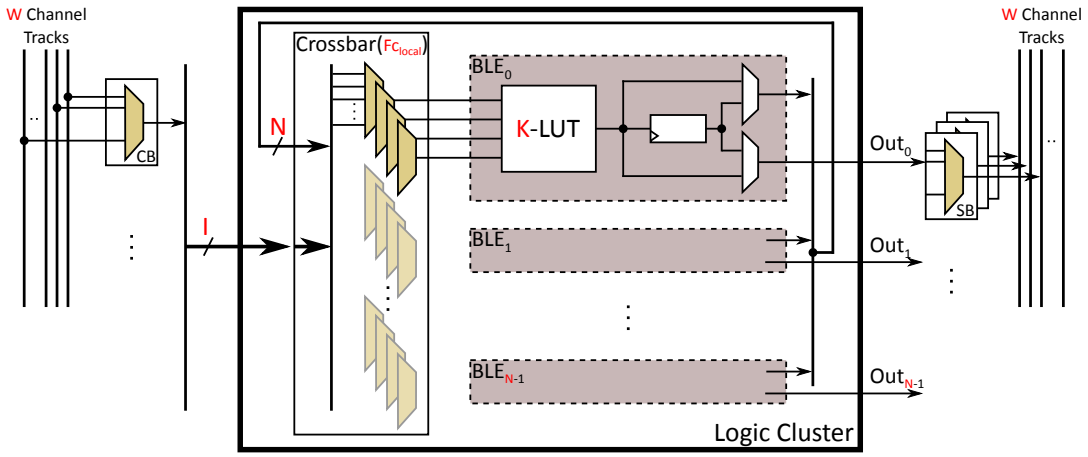


Figure 2.2 – The general structure of the FPGA's cluster architecture and its design parameters (K, N, etc.).

2.1.2 Fracturable LUTs

Fracturable LUTs are a key feature in modern FPGA architectures [Lewis et al., 2005] [Xilinx] that leverages the advantages of small-sized LUTs in architectures built with relatively large LUTs.

In general, a fracturable K-LUT is designed using two $(K-1)$-LUTs and additional multiplexers. Figure 2.3 shows an example of how a fracturable 6-LUT is built in a BLE with 8 inputs and two outputs. This fracturable LUT can operate either as (i) a single LUT with six inputs or (ii) two LUTs with five inputs each. Since the BLE has only 8 inputs, when operating in the two 5-LUTs mode, the LUTs have to share two of their inputs. By construction, to form a 6-LUT, the two 5-LUTs need to have the same inputs. So, the non-shared pins are assigned, by the routing, the same signals in both LUTs. Then the output multiplexer selects, depending on the value of the 6^{th} input, the output of either of the two 5-LUTs. To free one of the BLE inputs to deliver the 6^{th} signal, an additional multiplexer is added at the inputs. It provides the option of hard-wiring a third shared input between the LUTs.

The CAD tools are not usually exposed to the details of the design and the hardware implementation of the fracturable LUTs. All they require is sufficient information on its modes of operation and input/output connections. However, when actually modeling the architecture at the hardware level, all the implementation specifics are needed to correctly account for any overhead introduced by the additional multiplexers.

2.1.3 Depopulated Crossbars

The crossbar is one of the main routing elements in the cluster; it connects the cluster inputs (I) and the local feedback signals (N) to the LUTs. It also helps reduce the global routing effort

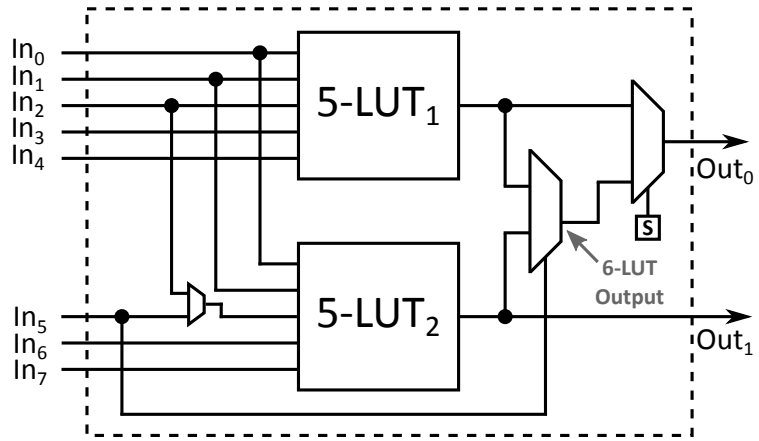


Figure 2.3 – An example of a fracturable 6-LUT in a BLE with 8 inputs and 2 outputs.

and congestion by handling the signal distribution within the cluster.

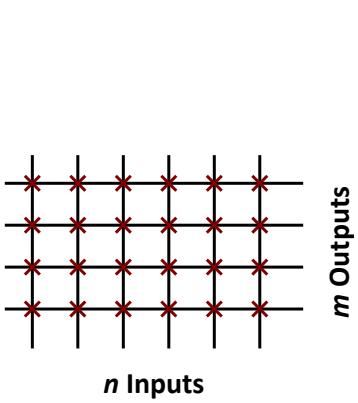
In general, an $n \times m$ crossbar is used to connect n inputs to m outputs as shown in Figure 2.4, where typically $n \geq m$. A programmable switch known as a *crosspoint* is used to connect an input wire to an output wire. The number of crosspoints in a specific crossbar determines its *population p*, while its *capacity c* is specified by the number of signals being routed through that crossbar. Crossbars are typically implemented as a set of m multiplexers where, again, m is the number of crossbar outputs. The size of the multiplexer depends on the number of connected inputs (i.e., crosspoints) per output. In the case of the architecture of Figure 2.2, the crossbar consists of $N \times K$ multiplexers where the size of each multiplexer is determined by the density of the crossbar ($F_{c_{local}}$) along with I and N.

As the clusters get bigger, the area cost of the crossbar can become very expensive. Thus, when the area of the architecture is critical, highly routable yet sparsely populated crossbars are preferred [Lemieux et al., 2000; Lemieux and Lewis, 2001].

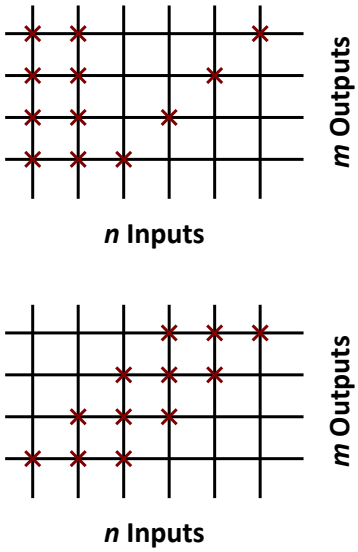
Three main types of crossbars can be highlighted.

Full crossbar

A *full crossbar* has a programmable switch at the intersection of every input wire with every output wire, which allows it to connect any input to any output giving it full flexibility. Full crossbars are also know as *fully-populated crossbars* since their population p is maximal, which is equivalent to an $F_{c_{local}} = 1$.



(a) Full crossbar



(b) Minimal crossbars

Figure 2.4 – $n \times m$ crossbars having n input wires and m output wires. (a) The full crossbar has a programmable switch at every intersection of the input and output wires, supporting all possible connections. (b) Minimal crossbars (two examples) allow any set of m inputs to be connected to all outputs but without any control on which input is connected to which output.

Full-capacity minimal crossbar

A *full-capacity minimal crossbar* (also known as *minimal crossbar*) has less flexibility than a full crossbar since it uses fewer switches. However, the minimal crossbars maintain the *full-capacity* property of the full crossbars since they can connect as many signals as the number of outputs in that crossbar. This means that any set of m inputs (out of n) can be connected to all m output wires. The main behavioral difference, when compared to full crossbars, is that minimal crossbars cannot always connect a specific input wire to a specific output wire. They do not provide full flexibility in the permutation of inputs/outputs, since some specific connections would not be feasible. A minimal crossbar always uses $p = (n - m + 1) \cdot m$ switches. Removing any additional switch forces the crossbar to lose its full-capacity property. Both full and minimal crossbars are referred to as *perfect* crossbars since they maintain full capacity.

Sparse crossbar

A sparsely populated crossbar is referred to as a *sparse crossbar*. Although there is no common convention on the degree of sparsity required to build a sparse crossbar, Lemieux et al. [2000] assume that a crossbar is sparse if it contains fewer switches than the minimal crossbar (i.e., $p \leq (n - m + 1) \cdot m$). This means that a sparse crossbar can never be perfect (i.e. can never achieve full capacity).

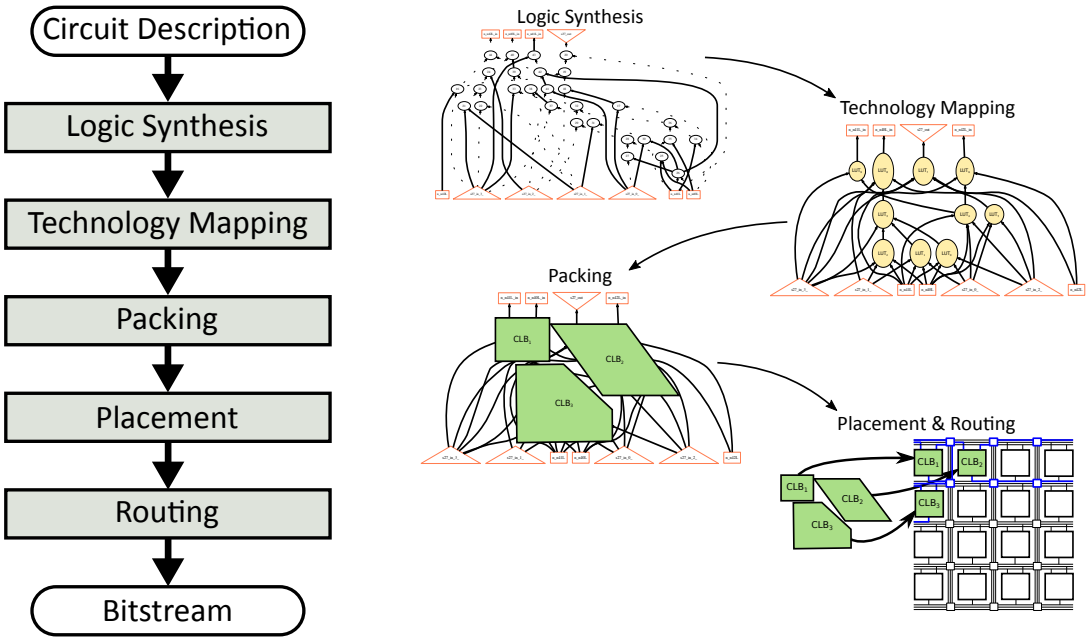


Figure 2.5 – The FPGA CAD flow.

Commercial FPGAs based on the Altera Stratix family use 50% sparse crossbars [Lewis et al., 2003] in their clusters, while Microsemi has FPGAs with multi-level crossbars, some of which are highly depopulated, reaching a 25% density [Greene et al., 2011]. It was also observed through academic research that low-density crossbars are beneficial for some specific architectures (e.g., $K = N = 6$) [Lemieux and Lewis, 2001].

2.2 FPGA CAD Flow

To implement any particular application on the FPGA architecture, the circuit has to go through the different phases of the FPGA CAD flow, shown in Figure 2.5.

Reading in a description of the circuit, usually in an HDL language, the flow starts first by synthesizing the circuit and converting it to a specific internal representation. The state-of-the-art academic tools, like the synthesis and verification tool ABC [ABC], synthesize the circuit into an And-Inverter Graph (AIG), which consists of a network of 2-input AND gates and inversions, before optimizing it, usually for delay.

The technology mapping phase then (i) identifies subgraphs of the network that can be implemented by the basic logic element of the FPGA (typically an LUT), and selects a particular set of subgraphs that can cover the entire circuit while minimizing some metric of interest, such as the critical path delay or the number of logic elements used.

At the end of the technology mapping, the circuit becomes a network of logic cells ready to be

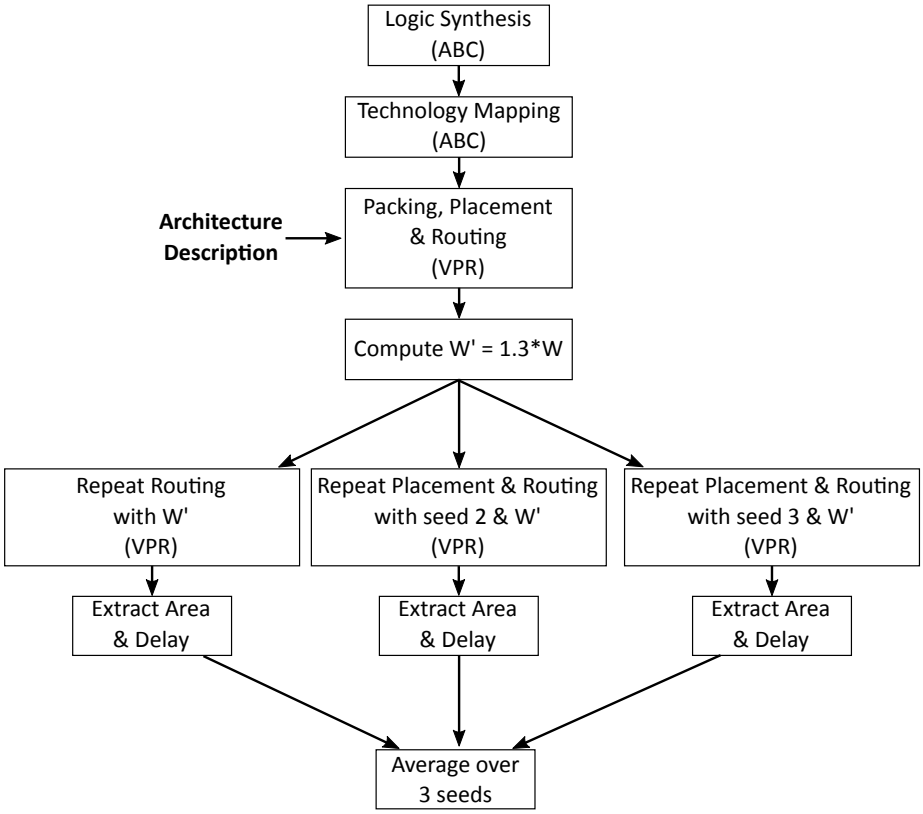


Figure 2.6 – The different steps of the experimental setup.

packed into the logic clusters. The packing algorithm decides on the logic cells that should be grouped together to form a cluster, typically guided by some metrics that vary, depending on the packing algorithm and optimization objectives.

The packed clusters are finally placed within the FPGA grid and all the routing channels and connections used to connect these clusters are determined by the router. Knowing exactly where the logic cells are, which functions they implement, and how they are connected, allows us, at the end of the CAD flow, to extract all the configurations of the SRAM cells and generate the respective bitstream, which will then be used to configure the FPGA and implement that particular circuit. Academic CAD tools, including the VTR flow, do not usually generate the bitstream but, instead, they simulate the overall architecture performance for the given benchmark.

2.3 Experimental Setup and Benchmarks

To evaluate the modeled architectures, we use the Verilog-to-Routing (VTR) project [Rose et al., 2012] which consists of a collection of tools and algorithms that handle the different phases of the CAD flow.

In all our experiments, we use the latest release of the tool, VTR 7.0 [Luu et al., 2014a], and provide it with a description of the modeled architecture in the XML format. As shown in Figure 2.6, for every benchmark, ABC [ABC] synthesizes and maps the circuit (previously elaborated using ODIN II) on K-LUTs, where K depends on the given architecture. Then, the mapped network is packed, placed and routed using VPR, with unlimited global routing resources. Knowing the minimum channel width (W) required to route the benchmark, we repeat the routing step but with a fixed channel width, set to 30% bigger than the detected minimum ($W' = 1.3 \times W$). The placement and routing stages are also repeated for three different placement seeds and the results are averaged over these seeds, in order to reduce the placement and routing noise. In the different stages of the CAD flow, the tools are set to optimize for the critical path delay.

We test our architectures on two sets of benchmarks: (i) the Big20 set of the MCNC benchmark suite [Yang, 1991] and (ii) the VTR benchmark suite [Rose et al., 2012]. The selected MCNC benchmarks consist of 20 large combinational and sequential circuits. However, the VTR benchmarks were more recently introduced as bigger circuits that can represent real applications and use memory and DSP blocks. We will, in Chapter 6, evaluate the effect of the benchmark selection on the conclusions of our architecture explorations.

3 Library Generation of Macrocells with Different Drive Strengths

The main objective of FPRESSO is to model FPGA architectures while abstracting the complexity of the transistor-level optimizations from the user by splitting the process into two different and completely separate phases: a library generation phase and a circuit optimization phase.

The approach relies on a library of all the macrocells that can be used to build an FPGA architecture, prepared during the library generation phase. This library contains the timing and area characteristics of multiple instances of every cell, each with a different drive strength. Built only once per technology, offline, this library is used, at runtime in the circuit optimization phase, to model the user-defined architectures. The user is only exposed to the architecture optimization phase, which is managed entirely by automated scripts and a standard-cell optimization tool, thus, requiring no particular user expertise.

This chapter focuses on the library generation phase and the different stages of the process, shown in Figure 3.1, as well as the challenges faced in building the flow using existing, off-the-shelf tools. Although the concept is inspired from standard-cell design, the notion of *cell*, in this context, is rather different from what is usually encountered in standard-cell libraries. Instead of dealing with gates and relatively small-sized cells, the library is built of intuitively-defined logic and routing elements that can be found in an FPGA architecture. These can range from small-sized cells that can be found in standard-cell libraries like inverters, flip-flops, and multiplexers, to relatively large elements like look-up tables and two-level multiplexers. The challenges of building our library come from its two main steps: (1) automatically size each cell and create a variety of versions with different drive strengths, and (2) characterize each of these cells to build a primed and comprehensive library.

3.1 Leveraging State-of-the-Art Transistor-Sizing Tools

For simple components such as standard cells, the sizing problem is not a terribly complex optimization problem as most cells are composed of just a handful of transistors. However, for complex components such as LUTs with tens of transistors to size, it becomes quite

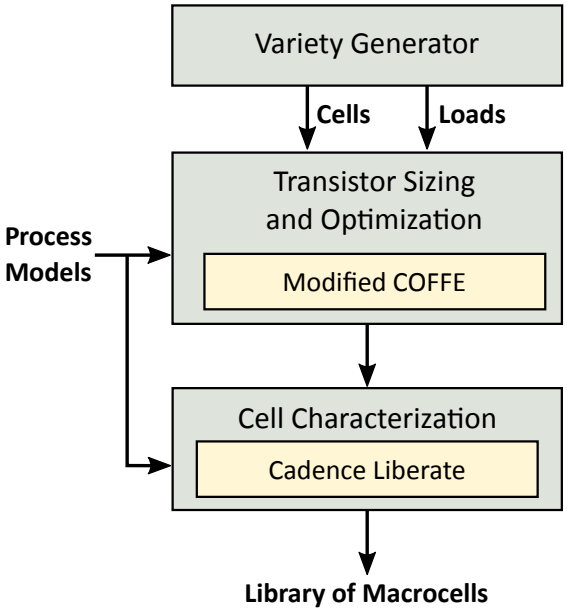


Figure 3.1 – The library generation flow.

challenging, even for an expert circuit designer, to decide the optimal transistor sizes and automation of the process is essential.

Luckily, as mentioned in the Introduction, automatic transistor-sizing tools for semicustom designs have been ported to reconfigurable circuits and FPGA architectures, in particular. Thus, instead of building our own transistor-sizing tool, we decide to leverage the state-of-the-art automatic FPGA transistor sizing tool, COFFE [Chiasson and Betz, 2013], and customize it to automatically size the cells and interface with our library generation flow.

In this section, we will first discuss COFFE's key features and approach to address the transistor sizing problem. We will then highlight the modifications we made to the tool so that it sizes individual cells and interfaces correctly with our library generation flow.

3.1.1 COFFE's Architectural Structure and Optimization Strategies

COFFE [Chiasson and Betz, 2013] is designed to automatically optimize the transistor sizes of a parametrized standard FPGA architecture. Figure 3.2 shows the tile architecture supported by COFFE and the parameters through which the user can customize the architecture to be modeled.

COFFE uses SPICE simulations to iteratively search a range of transistor sizes and to find the optimal combination for a given optimization criteria. An exhaustive exploration of all transistor sizing combinations is not feasible given the very large search space and the need for time-consuming SPICE simulations. So, COFFE reduces the complexity of the transistor

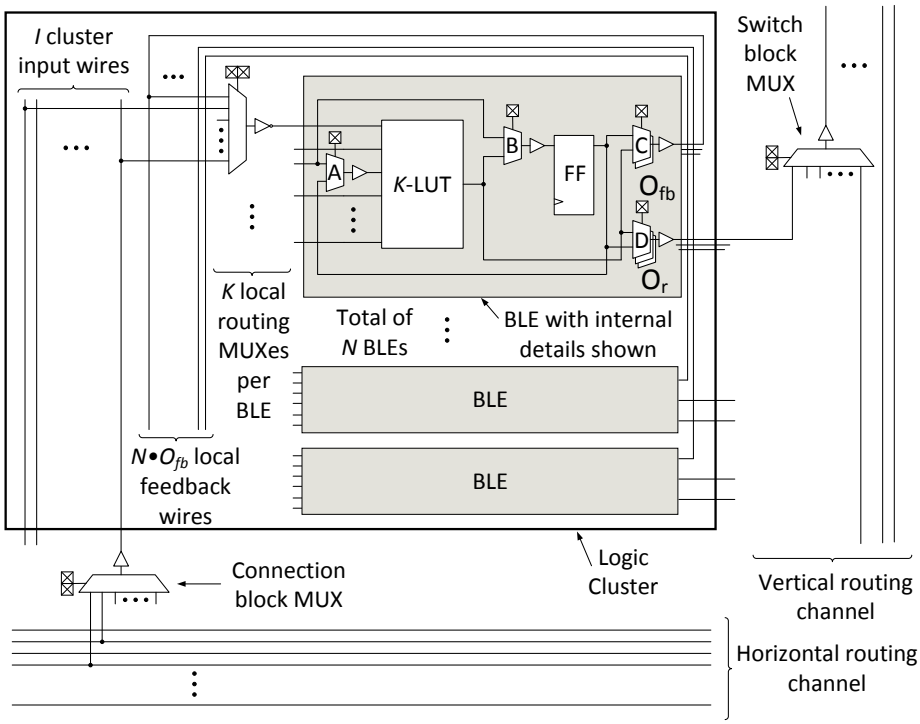


Figure 3.2 – The FPGA tile architecture supported by COFFE [Chiasson and Betz, 2013].

sizing problem by exploiting the reconfigurable nature of the FPGA. It tries to exploit the abundant symmetries in the transistor netlist of each cell to reduce the number of transistors to be sized. For example, all the paths from one SRAM cell to the output of an LUT are identical and will have the same transistor sizes, hence, only the transistors of a single path are sized. To further speed up the sizing process, the sizable set of transistors is split into several groups of a controlled number of transistors (usually about 5 or 6 transistors). Using this divide-and-conquer approach, the number of transistor sizing combinations to be explored is further reduced. And to compensate for any side effect of this transistor clustering approach, COFFE iteratively sizes the transistor groups until no further improvement is obtained or after reaching a maximum number of iterations. According to their experiments, the authors show that COFFE usually converges to a solution after 2 to 4 iterations [Chiasson and Betz, 2013].

However, the programmability of the FPGAs means that the critical path is application dependent, making it hard to determine the critical path at design time while optimizing for delay. COFFE implements two approaches to deal with this issue. In the first approach, COFFE constructs a representative critical path that includes one instance of each cell and computes the delay as a weighted sum of the delays of these cells, where the weights are determined based on the frequency of occurrence of each cell on the critical paths of known benchmarks. The second approach to deal with the lack of a clear critical path is to simply optimize each cell individually.

To evaluate and rank a particular combination of transistor sizes, COFFE uses a weighted area-delay product as a cost function, in the form of

$$Cost = area^a \cdot delay^d, \tag{3.1}$$

where a and d are user-defined parameters that prioritize area over delay, or vice-versa. The circuit delay for a transistor sizing combination is estimated using SPICE simulations. The area, however, is calculated by component-dependent formulas that are programmatically encoded in the tool. At the end of the optimization process, the transistor sizing combination that achieves a minimum cost is selected.

COFFE models not only the long wires like the local interconnects between the cluster blocks, but even the short metal that connects two transistors within a single component. The authors emphasize the importance of modeling all wires, even the shortest connections, and its impact on the overall delay and area.

With all these characteristics, COFFE happens to be very suitable for the transistor sizing step needed in our library generation phase. Its level of automation and its parameterizable FPGA architecture provide us with a reliable infrastructure that can be used, with some modifications as detailed in Section 3.1.2, to build a customized transistor-sizing tool that satisfies the requirements of our approach.

3.1.2 Modifying COFFE to Size Individual Cells

As an automatic transistor sizing tool, COFFE happens to have all the features needed to size our macrocells. The tile architecture that COFFE supports (shown in Figure 3.2), is an excellent starting point, since it already contains all the cells required to build a standard, state-of-the-art FPGA.

However, COFFE is designed to size and model the entire FPGA architecture, while we are interested in individual cells out of that architecture. Thus, we modify COFFE to isolate and size each component separately. These components, which will form the cells of our library, are: the K-LUTs, two-input multiplexers, variable-input two-level multiplexers, and flip-flops (although COFFE does not actually size the flip-flops). Since each component is originally sized within the context of the architecture that is generally being modeled, the load at the output of that component is naturally and automatically determined, for each architecture, by the circuit it is driving. However, when the component is isolated, it is sized separately without any circuit to drive. In other terms, at the time when the component is sized, we do not know in which architecture it will be used and, by that, what circuit and load it will be driving. So, one of the main modifications imposed on COFFE is to take as input a capacitance to be used as an output load during the transistor-sizing process. This allows it to correctly size the components and to generate a variety of cells with different drive strengths, as will be discussed in Section 3.2.1.

By optimizing a single component at a time, the critical path of that circuit is easily identified, making the second optimization approach of COFFE—the one that optimizes each circuit individually—highly convenient for our objectives. We also keep the same weighted area-delay product cost function of Equation 3.1 to evaluate the different solutions. As mentioned in Section 3.1.1, COFFE models the wire loads even for the smallest connections between the transistors. This implies that the wires within our cells are inherently automatically modeled during the transistor sizing process.

The modifications to the scripts of COFFE also include extensions to support components with larger number of inputs. For instance, the biggest LUT it originally supported was a 6-input LUT with the netlist generation of each *K*-LUT hard coded, individually, in the scripts. So, we extend the scripts to support LUTs with an exceedingly large number of inputs, and generalize the process to automatically generate the SPICE netlist and insert the internal buffers. Although we currently limit the cells of our library to the components that can be extracted from COFFE, new components can be easily added. Once we know the transistor design on the component to be added, it can be coded into COFFE to (1) generate automatically its SPICE netlist, (2) determine the path(s) and transistors to be sized, and (3) break these transistors into groups that can be iteratively sized. In general, once the final transistor sizes of a component are determined, we add a step into COFFE to generate the complete SPICE netlist by duplicating the path that has been sized, depending on the symmetries of each component.

3.2 Generating Variety of Cells

In a standard-cell based ASIC design, a rich library with a wide range of drive strengths is essential to approach full-custom design efficiency [Chinnery and Keutzer, 2002]. Similarly, for our library of cells we provide a wide range of different sizes, sweeping from very small components optimized for small loads, up to significantly large ones dimensioned to drive heavy loads.

3.2.1 Variety Through Output Loads

In our modified version of COFFE, we isolate each component from its surroundings and size it for a certain load capacitance C_L. To generate multiple drive strengths per component, we vary the load capacitance over a representative range of values. We use an exponential distribution of the form

$$C_L(n) = C_{\text{Inv}} \cdot 2^{\frac{n}{2}}, \tag{3.2}$$

where C_{Inv} corresponds to the input capacitance of a minimum-width inverter and n the load index, with $n = 0,1,2,...N$ and the maximum load index defined by N. Accordingly, $C_L(n)$ ranges from a minimum value equivalent to the input capacitance of a minimum-width inverter, in the selected technology, to a limited maximum value for which the load is

considered exceedingly large. This maximum value is equivalent to a load for which the sized component will never be used by the synthesis tool during the architecture optimization phase. With this, we hope to provide a representative set of drive strengths for every component. However, we realize that changing only the output load does not provide a sufficient variety of cells, as will be shown in Section 3.2.2.

3.2.2 Variety Through Optimization Objectives

We understand that a rich library should include a large variety of drive strengths for each cell. This variety should be evident from the cell's output driver, whose size must vary significantly from a minimal size ($\approx 1\times$) to significantly larger sizes ($> 100\times$). This is due to the fact that the output driver is mainly responsible for restoring the output signal and delivering enough drive strength for the output load.

Experimenting with different weight values (a and b) in the optimization cost function (Equation 3.1), we realize that a single combination of weights (e.g., $a = 1$, $d = 1$) is not enough to create the type of variety we want in our library. However, generating a large set of transistor sizings with different optimization weights coupled with a later pruning can create the desired variety.

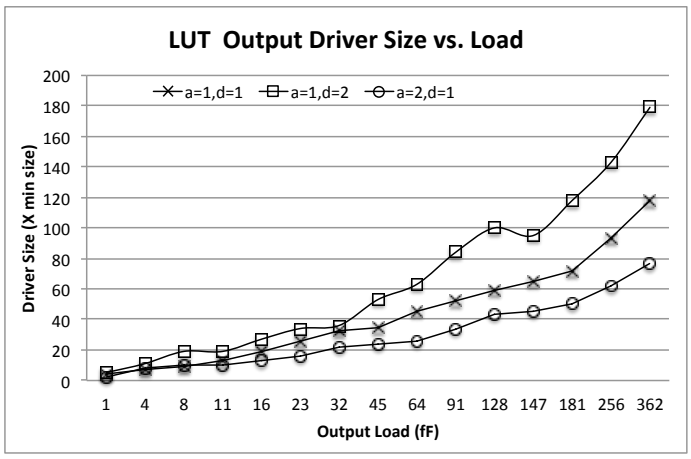
Figure 3.3 shows the size of the output driver of several cells, for different loads. A balanced optimization ($a = 1$, $d = 1$) does not generate enough variety, and in some cases it stops increasing the size of the output driver way too early. This happens when the area increases by a factor that is larger than the delay reduction. Accordingly, an optimization that favors delay but that does not completely ignore area (e.g., $a = 1$, $d = 2$) will generate larger drive strengths. At the same time, an optimization that gives more weight to area (e.g., $a = 2$, $d = 1$) will not generate large drive strengths but components that may still be useful to optimize the area in non critical paths of the FPGA architecture.

Hence, we use the modified version of COFFE to automatically size the different cells needed to build the FPGA architecture, and rely on these two approaches to generate the required drive-strength variety for each cell.

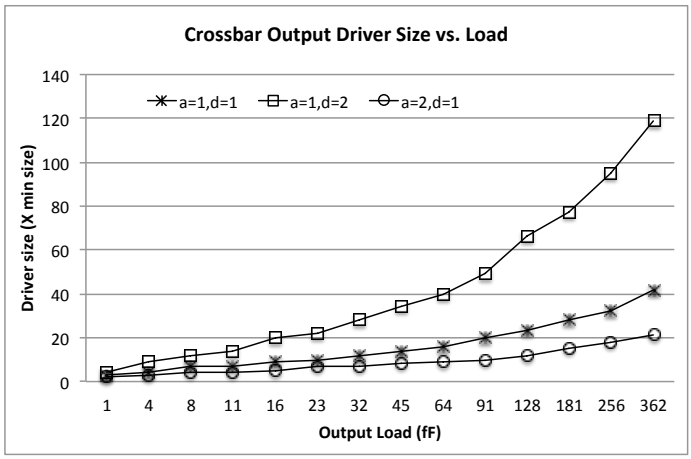
3.3 Cell Characterization

Having the transistor sizing and optimization step automated and all the cells of interest optimized, with different drive strengths, the next step is to provide measurements (e.g., input-to-output delay) for each cell, under various conditions. The process is known as *cell characterization* and the conditions are either specified by the designer or constrained by the used technology.

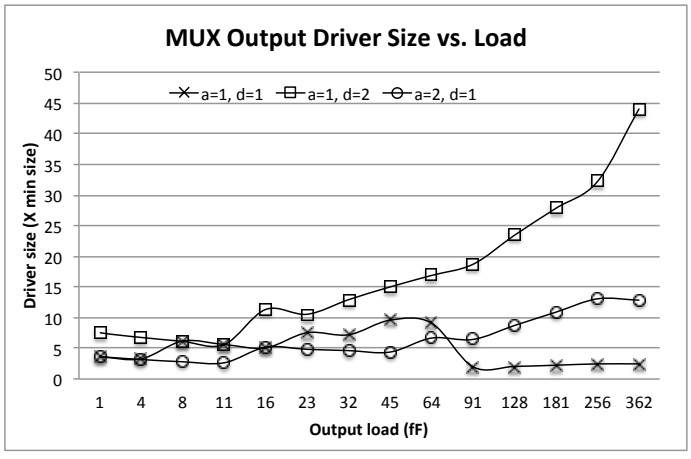
Cell characterization is a very well understood process: conceptually, SPICE-like simulations are run within some simple testbenches with varying driving slopes and load capacitances.



(a) 5-LUT output driver size.



(b) 25:1 2-level multiplexer output driver size.



(c) 2:1 multiplexer output driver size.

Figure 3.3 – Analysis of the output driver size for different cells, optimization criteria and output loads.

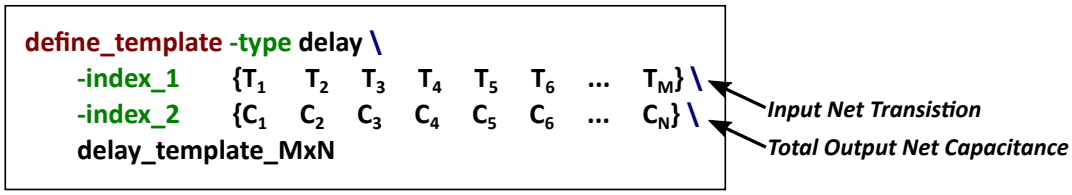


Figure 3.4 – The characterization delay template. To characterize a cell, a delay template is defined with a list of values for each of the two metrics that influence any input-to-output delay: (1) the input net transition and (2) the total output net capacitance.

Table 3.1 – The library's delay matrix. The measurements of every input-to-output rise/fall delays are stored in a two-dimensional matrix, indexed by the characterization metrics (namely, the input net transition and the total output net capacitance).

		Total Output Net Capacitance				
		C_1	C_2	C_3	...	C_N
Input Net Transition	T_1	$Delay_{11}$	$Delay_{12}$	$Delay_{13}$	...	$Delay_{1N}$
	T_2	$Delay_{21}$	...	...	...	...
	T_3	$Delay_{31}$	...	...	...	...
	...	...	...	...	...	...
	T_M	$Delay_{M1}$	$Delay_{M2}$	$Delay_{M3}$	...	$Delay_{MN}$

Process corner, temperature, and voltage can also be added as variables but this is outside the scope of our goals. EDA tools that perform the task in a completely automated way exist and, of course, they interface very well with other parts of the semicustom toolchain. To characterize our cells and build our library, we decide to use *Cadence Virtuoso Liberate*: an advanced library characterization tool that creates electrical views, such as timing and power, in standard formats like the *Synopsys Liberty format (.lib)*.

To characterize a cell, Liberate takes as input the SPICE netlist of the cell along with the foundry device models. In our case, we take the SPICE netlist generated by the modified version of COFFE as explained in Section 3.1. We also generate characterization scripts for Liberate in which we specify the characterization conditions and parameters. They span from operating conditions, such as the exact temperature and voltage, to specific tuning parameters. We also provide Liberate with a delay template, as shown in Figure 3.4, that lists the values of the two metrics for which the cell is to be characterized: *input net transition* (also known as input slew) and *total output net capacitance*. Liberates assigns a value from each list to the input

```
add_cell_attribute {cell_name} {
    area : A;
}
```

Figure 3.5 – The area of each cell, as added to the characterization scripts.

transition time and output capacitance of the cell in question, respectively, and then measures every input-to-output rise/fall delay. This is repeated for all the combinations of these two metrics and stored in the library. Table 3.1 shows how these measured delays are stored in the library and indexed by the two metrics, assuming N values for the total output net capacitance and M values for the input net transition metric. Thus, the library consists of such matrices, that detail the rise and fall delay from every input to every output of the characterized cell. In addition to the input-to-output delays, Liberate also computes and stores in the library the output net transition times, which will be used during the circuit synthesis and optimization phase to estimate the input net transition times of connected cells.

The areas of the cells are actually determined once they are sized and they are provided as one of the outputs of the modified COFFE version. But to include all cell characteristics in the same library, we need to provide Liberate with these areas. Figure 3.5 shows how the area of every cell is added to the characterization scripts so that it is included in the final library file. Although we might be interested in the future in modeling, on top of the delay and area, the power of each cell, we currently disable the power and leakage features in Liberate. Using these features with our cells' level of granularity and complexity presents a major challenge to Liberate, to a point where it does not even terminate in reasonable time. Thus, we disable this feature for now, but, if we want to include power characteristics in the future, we need to find ways to reduce the complexity for the characterization tool. We actually face similar complexity problems even in measuring the delay and determining the cell functionality, but we were able to leverage the FPGA's symmetries and reconfigurability to circumvent these challenges, as detailed in Section 3.4.

3.4 Challenges in Characterizing Complex Cells

Our components are qualitatively similar to standard cells and, in some cases, are practically identical, such as flip-flops and 2-to-1 multiplexers. Unfortunately though, many of the critical components, such as LUTs and large two-level multiplexers, are much bigger than the biggest typical standard cells, in terms of both transistor count and number of inputs. This implies that, for understandable scalability issues, a tool like Liberate naturally fails to characterize some necessary components. We will discuss in this section how we use the cell symmetries and reconfigurability to circumvent these problems.

3.4.1 LUT Characterization

Liberate is capable of characterizing small size LUTs, such as LUTs with 2 or 3 inputs. However, as the number of inputs increases, the number of SRAMs grows exponentially, making the LUT characterization a major challenge for Liberate. Having a complex functionality and a high number of variables to track, Liberate fails to characterize the cell.

In order to overcome the problem and simplify the task for the characterization tool, we reduce the complexity of the LUT by exploiting the symmetries in its transistor design. As seen in Figure 3.6, a signal can travel from an input (e.g., in_A) to the output through what seems to be different paths. In practice, however, these paths include transistors of identical sizes because each transistor belonging to a level of the binary tree has the same size. The only actual difference is whether the input inverter is used or not. Thus, it is sufficient to pick a limited number of *representative paths* during the characterization to cover all the different input-to-output timing arcs, and this can be done by fixing the configurations of the SRAMs (i.e., setting them to '1' or '0'). These representative paths must be selected in a way that guaranties the characterization of both inverted and non-inverted input-to-output paths. Furthermore, these paths must enable all possible signal transitions (i.e., rise-fall, rise-rise, fall-fall and fall-rise) between the inputs and the output. For that purpose, we set the configuration bits S_0 and S_7 in the example of Figure 3.6 to '1' and the remaining ones to '0'. By configuring the 3-LUT in this way, all the input-to-output timing paths, along with their respective rise and fall delays, will exist in the generated library. To configure the LUT, the SRAM cells are connected directly to *VDD* and *GND*, reducing the number of variables visible to Liberate.

Once the LUT is characterized for the representative paths, the SRAMs are then added back as input variables and the library is corrected with the respective Boolean function. It is important to bring back the complete LUT interface into the library so that it can be directly associated with its circuit description when read into the later stages of FPRESSO. Also, the synthesis and optimization tool used in these stages identifies the different sizes of the same cell (i.e., with different drive strengths) only if they have the same logic function and if it uses all input variables.

3.4.2 Two-Level Multiplexer Characterization

FPGA crossbars are generally designed as 2-level multiplexers [Lewis et al., 2005], as shown in Figure 3.7. Each multiplexer is usually connected to all or to a fraction of the crossbar inputs, depending on the desired sparsity. As mentioned in Section 2.1.3, the crossbar of the Stratix-IV FPGA, for instance, has 72 inputs and is half populated [Lewis et al., 2005], [Lewis et al., 2009], which means that each 2-level multiplexer has 36 inputs and 12 SRAMs.

Hence, similar to the LUTs, these multiplexers can have a large number of inputs, making it impossible for Liberate to characterize. However, to ensure the multiplexer functionality, only

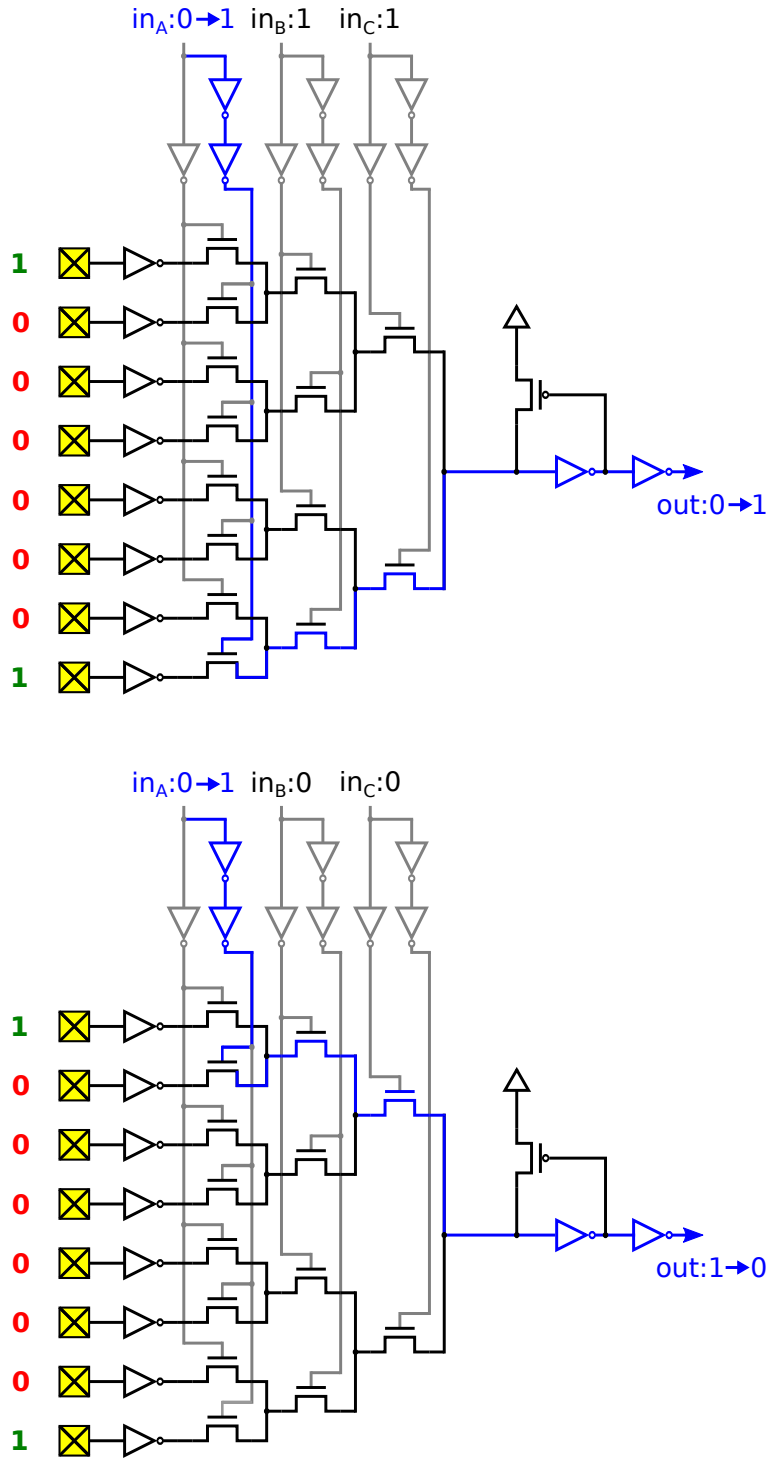


Figure 3.6 – The transistor design of a 3-input LUT. Representative paths that ensure all signal transitions between the inputs and the output are selected to simplify the characterization of an LUT.

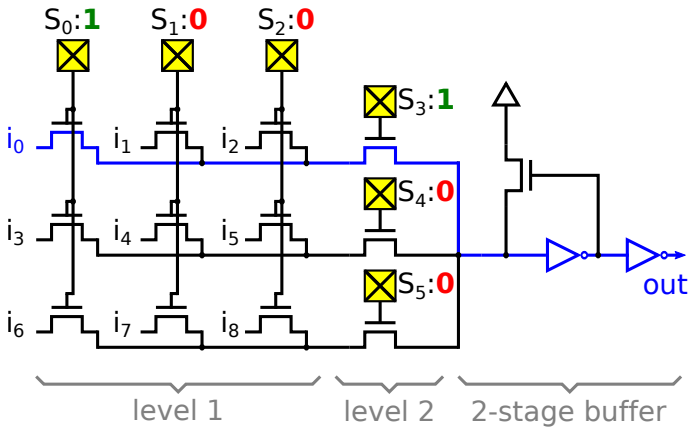


Figure 3.7 – The transistor design and configuration of a 2-level multiplexer. This multiplexer has 9 inputs and is called a 3×3 mux since it has 3 SRAMs in each level.

two SRAMs are set during one configuration (one SRAM in the first level and one in the second) to connect a single input to the output. This property is used to simplify the characterization of the 2-level multiplexers by characterizing configured versions of the multiplexers, limiting the number of transistors and reducing the complexity of the process. For each input, the multiplexer is configured to enable the path from that particular input to the output by connecting the two related SRAMs to *VDD* and the remaining ones to *GND*. Figure 3.7 shows the general structure of a 9-input 2-level multiplexer, where, to connect input i_0 to the output, S_0 and S_3 are set to '1' while the remaining configuration bits are set to '0'. Once an input-to-output path is enabled, the remaining inputs are no longer relevant and can be forced to a logic value ('1' in this case). The multiplexer input is then characterized and the process is repeated for all inputs, which generates multiple library files. Thus, in a final step, all the generated libraries are merged back into one, the SRAMs are reintroduced as input variables and the logic function is corrected.

3.4.3 Switch/Connection Block Characterization

Within the FPGA tile, the switch Blocks (SBs) and Connection Blocks (CBs) are basically multiplexers used to connect the cluster outputs and inputs to the global routing. Since they have a large number of inputs, the SBs/CBs are designed as 2-level multiplexers, similar to the multiplexers of the crossbars.

Figure 2.2 shows the SB/CB connections to the cluster's outputs/inputs and to the routing channel. Given a single cluster, the inputs of the SBs/CBs are hard to identify since they could be outputs of other clusters, for example. Therefore, when modeling one single cluster, connecting all the inputs of the SBs/CBs is not feasible. And since the optimization tool used in FPRESSO requires all the inputs to be connected, the 2-level multiplexers created in Section 3.7 cannot be used.

Instead, we create a new set of 2-level multiplexers where only one input is available and can be connected to the output. This can be done by selecting and characterizing a single path from one input to the output. The process is similar to the one used in Section 3.4.2 where the single path is selected by setting two particular SRAMs to '1' and the remaining ones to '0'. The first input of the multiplexer (i_0) is chosen in this case, as highlighted in Figure 3.7. The multiplexers are characterized with only this configuration and the timing information between i_0 and the output is added to the library. To differentiate this special type of 2-level multiplexers, they are given a function with only i_0 and the selected two SRAMs (S_0 and S_3 in the case of the 3×3 multiplexer of Figure 3.7) as variables. Having a different number of inputs and a different function, this type of multiplexers will be used, in the optimization phase of FPRESSO, only for SBs and CBs.

3.5 Discussion

The tasks described in this chapter are essential to the preparation of the library of cells, used in the later stages of FPRESSO to model the FPGA architecture. As with standard cells, the library development is an elaborate and lengthy process, performed offline by fairly experienced designers. In case of porting the library from one technology node to another, the process is almost automatic, but the expertise of a transistor-level designer is required if one is to add new components to the library (for instance, non-LUT logic blocks). Once the library is available, architects can model new FPGA architectures by composing the circuits using the predefined functional blocks, as will be discussed in details in Chapter 5. They simply define, functionally, the architecture of the FPGA while completely ignoring electric and timing issues. FPRESSO automatically models and optimizes the described architecture, using the cell characteristics stored in the library, and returns, in the end, the same architecture fully annotated with the required delay and area estimations.

To explain and illustrate the library generation process, we use, throughout this chapter, specific designs and implementations of the FPGA cells. By relying on COFFE to automatically size the cells, we are inheriting the same design decisions adopted in COFFE. However, our approach itself is generic and can support any transistor-level design of the cells. Furthermore, being able to define and identify individual cells, FPRESSO can support multiple designs of the same cells, allowing for a wider variety of implementations. For example, the LUT design of Figure 3.6 is not a unique way of implementing the LUT, thus, various implementations can be added as different cells and used to build the FPGA, depending on the user's specifications.

An important stage of FPRESSO's modeling is to account for the wires within the architecture. As stated in Section 3.1, since COFFE already models the wires that connect the different transistors within the cell and takes their parasitics into account while sizing the cell, all the wires within our library cells are inherently modeled as well. However, we still need to model the wires that connect the cells to each other, in the architecture. Since our task is to simply model the delay and area of the architecture, we do not have any notion of how the

circuit layout might be, making it hard for us to determine, at this stage, the length of those wires. To address this problem, FPRESSO relies on a floorplanning algorithm that optimizes the placement of the cells to reduce the total wire length and to model the wires connecting them. So, we will first explain, in Chapter 4, this wire modeling approach before covering, in Chapter 5, the remaining stages of FPRESSO's main flow. .

4 Automatic Wire Modeling of FPGA Architectures

Modeling the wireloads is an essential step towards a correct and comprehensive modeling of an FPGA architecture. COFFE, for instance, emphasizes on the importance of accounting for all the wires within the modeled circuit, even the smallest ones connecting individual transistors [Chiasson and Betz, 2013]. The authors study and highlight the impact of wireloads on the modeled area and, most importantly, delay of the architecture, where the optimized cluster path almost doubles in delay after including the wireloads.

In FPRESSO, a part of the wire modeling is already performed, albeit indirectly, in the library generation phase. By sizing our library cells using the customized version of COFFE, we are automatically modeling the wires that connect the different transistors within those cells. We still need, however, to account for the longer wires used connect the cells and which, arguably, have a bigger impact on the overall model.

The difficulty of the wire modeling problem is in determining the length of the wires at this early stage of the flow. Ideally, a complete circuit layout determines the relative placement of the different blocks of the architecture and, hence, the length of the wires connecting them. However, with the level of abstraction and efficiency required from the modeling tools, a quick and easy estimation of the wireload effect is essential. COFFE leverages the fixed structure of the architecture it supports and uses a simple topological order to place its blocks and estimate the length of the wires. This approach though cannot be applied to FPRESSO since the architectures to be modeled are only known at runtime and can have different structure and characteristics.

In this chapter, we present a method to automatically determine the length of the interconnects within the FPGA cluster, generically, and irrespective of its structure or the components used [Zgheib and Ienne, 2016]. The approach itself is modular and can be incorporated into modeling tools or used independently. However, in our case, we use it as a fundamental step in FPRESSO, to estimate the wireload effect while modeling the FPGA architecture.

Researchers have looked into the wire modeling problem in the past, whether for generic

circuits [Stoobandt, 2001] or analytical models for FPGA architecture [Smith et al., 2009]. Perhaps the most related research to our work is GILES [Padalia et al., 2003], a tool that generates transistor-level schematics and presents a place and route algorithm to obtain a compact layout of the architecture. However, our goals are fundamentally different since we do not plan on generating layouts from our design, but we target a fast modeling and estimation of the delay/area of an FPGA architecture, while taking into account the parasitics of the wires within that architecture.

4.1 The Wire Modeling Problem

To model the wires and measure their length, the dimensions of the different cluster components must be known as well as their placement within the cluster. And, to minimize the wireload, all the components connected through the same net must be placed as closely as possible to reduce the length of that net. So, the architecture can be seen as a network of components that must be placed within the cluster, as densely as possible. And by that, the wire modeling problem can be transformed into a floorplanning problem with the optimization objective of minimizing the wirelength.

Floorplanning is a well studied problem in VLSI design [Sherwani, 1995] and usually optimizes the placement for area and/or wirelength. There exist many floorplanning optimization algorithms, the most common of which are based on the *simulated annealing* heuristic [Kirkpatrick et al., 1983]. Generally, the algorithms start by representing the circuit using a graph or a tree of nodes before performing graph manipulation and optimization. Some of the earliest algorithms use normalized Polish Expressions [Wong and Liu, 1986], sequence pair [Murata et al., 1995] or O-Tree representations [Guo et al., 2001, 1999]. However, a more recent representation, the B*-Tree, was introduced as an easier-to-manipulate alternative representation [Chang et al., 2000; Wu et al., 2003].

Our approach starts first by converting the cluster architecture, with all its components (referred to as modules in the floorplan) and connections (wires or nets) into a B*-Tree representation. Then a simulated annealing based algorithm is implemented to floorplan the tree and optimize the placement of each module in order to minimize the overall wirelength. Once the floorplan is optimized, the best solution is chosen as the final placement of all the cluster components and the length of each net is computed to derive its respective load and resistance.

Before explaining the details of the floorplanning algorithm in Section 4.3, we first introduce the B*-Tree representation and its characteristics, used to determine the location of the architecture blocks in the optimizes floorplan.

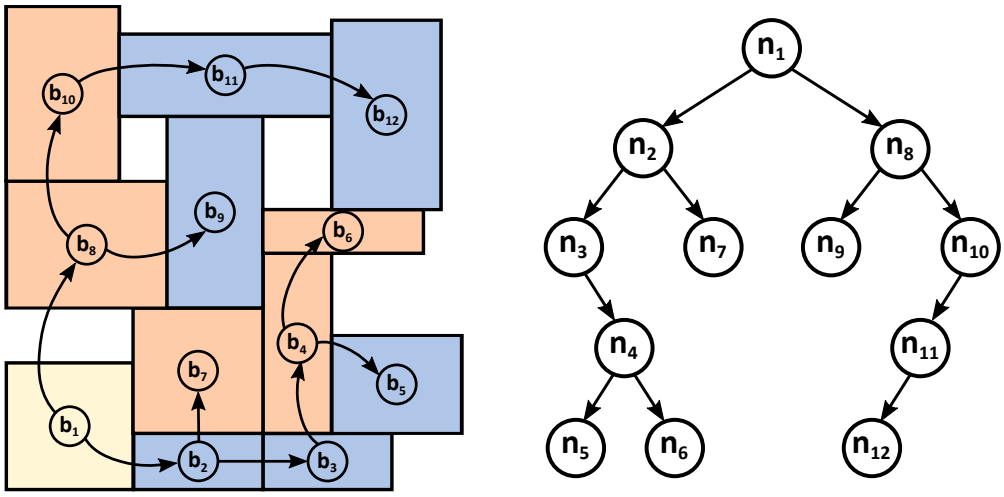


Figure 4.1 – An *admissible* floorplan, compact on the lower left corner, and its equivalent B*-Tree.

4.2 B*-Tree Representation

A *B*-Tree* is a representation in the form of a binary tree that was introduced to overcome the limitations of existing floorplanning representations [Chang et al., 2000; Wu et al., 2003]. A B*-Tree depicts the floorplan through a set of predefined geometric relationships. Each node (n_i) of the tree corresponds to a placed module (b_i) where the root of the tree is the module placed at the lowest left corner of the floorplan. The placement of each node is relative to the placement of its parent. If node n_j is the left child of node n_i, then b_j is placed on the right-hand side of b_i, directly adjacent to it. In this case, the x-coordinate of b_j is determined by $x_j = x_i + w_i$ where x_i is the x-coordinate of b_i and w_i its width. If n_j is the right child of node n_i, then b_j is placed immediately above b_i so that $x_j = x_i$. Figure 4.1 shows a B*-Tree and its equivalent floorplan, generated following the geometric relationships of the B*-Tree nodes.

To convert a B*-Tree to a floorplan, the tree is traversed in a way similar to the *Depth-First Search* by placing the root first at the lower left corner and then recursively tracing and placing its left subtree first then the right one. Similarly, an existing placement can be transformed into a B*-Tree if it is *admissible*, meaning that it is compact on the lower left corner in such a way that no module can be further moved left or down. The floorplan of Figure 4.1 is an example of an admissible placement. The B*-Tree is derived by starting from the module in the lower left corner (b_1), setting it as root to the tree and then recursively traversing the modules on its right, building the left subtree of the B*-Tree before visiting the module above it, building, also recursively, the right subtree.

Although the x-coordinates of the B*-Tree nodes are easily derived, the y-coordinates must be computed during its conversion to a floorplan. When placing a module b_i at x_i, it must not overlap with the existing modules, so its y-coordinate is determined by the maximum

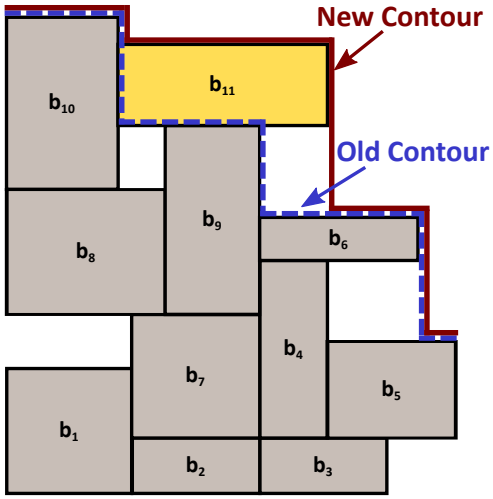


Figure 4.2 – The horizontal contour before and after adding module b_{11} to the floorplan.

height of the current floorplan between x_i and $x_i + w_i$. For that purpose, a *horizontal contour* of the existing floorplan is saved, keeping track of the maximum height of the modules already placed [Chang et al., 2000; Guo et al., 1999]. Figure 4.2 shows the horizontal contour before and after placing module b_{11}. It was shown that, using the contour, the y-coordinate of a newly inserted module is computed in $O(1)$ time [Guo et al., 1999].

4.3 Floorplanning Algorithm

Simulated annealing [Kirkpatrick et al., 1983] is commonly used to optimize the placement of floorplans in VLSI design [Sechen and Sangiovanni-Vincentelli, 1986a,b] and variations of the algorithm have been developed over the years to improve the runtime through parallelization [Chandy and Banerjee, 1996; Fang et al., 2009], modify the cooling process [Chen and Chang, 2006; Sechen and Sangiovanni-Vincentelli, 1986b], or change the cost function to target specific objectives [Adya and Markov, 2003].

In our wire modeling and floorplanning problem, our objective of minimizing the wirelength is simple and the number of modules to place is not exceedingly large, so a basic version of the simulated annealing algorithm should achieve the desired optimization.

The adopted algorithm starts by taking a randomly generated solution in the form of a B*-Tree. Each node of the tree is equivalent to a module in the floorplan or a component in the cluster architecture (e.g., K-LUT, 2:1 multiplexer, etc.). Having the node at a particular location in the tree is like specifying its (x, y) coordinates in the floorplan, as explained in Section 4.2.

As shown in Algorithm 1, starting with an initially high temperature T, the tree is perturbed M times, where each time the cost of the new tree is computed. If the new solution is better than the existing one, it is selected for the next iteration. However, if the new tree has a higher cost,

Algorithm 1 Simulated annealing algorithm

```
1:  BT = GenerateInitialRandomBTree()
2:  T = ComputeInitialTemperature()
3:  n = 0
4:  repeat
5:      m = 0
6:      repeat
7:          newBT = Perturb_BTree()
8:          Δcost = cost(newBT) − cost(BT)
9:          p_uphill = e^(−Δcost/T)
10:         if (Δcost ≤ 0) or (random < p_uphill) then
11:             BT = newBT
12:             if (cost(BT) < Cost(best_BT) then
13:                 best_BT = BT
14:             end if
15:         end if
16:         m++
17:     until (m ≥ M)
18:     n++
19:     T = λ^n T
20: until T < ϵ
```

the probability of accepting worse solutions, referred to as *p_uphill*, is computed and the new tree is accepted with a probability p_uphill. The uphill probability is defined by

$$\text{p\_uphill} = \text{e}^{\frac{-\Delta\text{cost}}{T}}, \tag{4.1}$$

where Δcost is the difference in cost between the new solution and the current one, and T is the current temperature. Since, initially, T is much larger than Δcost (as explained in Section 4.3.4), the probability of accepting a worse solution is very high, allowing for the search to escape local minima. As the temperature T decreases, p_uphill gets smaller so that it is less likely to choose a worse solution and becomes negligible as T tends to its final limit ϵ. p_uphill also depends on the value of Δcost in such a way that solutions with a larger cost difference are less likely to be chosen than the ones with a smaller cost difference.

After a certain number of iterations M, the temperature is decreased and the entire process is repeated but, consequently, with a smaller uphill probability. The best solution of all iterations is saved and returned as the final solution after the system freezes.

We will now elaborate on the main aspects and key features of the algorithm, namely the initial solution with which the optimization starts, the perturbation of the B*-tree to create new solutions, the cost function used to evaluate them, and the temperature that emulates the annealing behavior.

4.3.1 Initial Solution

The described simulated annealing algorithm takes an initial solution and iterates by perturbing it, in the search for the optimal one. This initial solution is a randomly generated B*-Tree.

The tree is built by first choosing a node (which represents a component in the architecture), at random, as the root of the tree, and then adding more randomly-selected nodes to the tree until it includes all nodes. Whenever a node n_j needs to be added, one of the existing tree nodes is selected at random to become its parent. If the parent node n_i is a leaf (i.e. a node without any children), n_j is randomly added as its left or right child. But if the parent n_i has only one child then n_j is added as its other child. Naturally, parents having already both children are not considered.

4.3.2 Perturbation

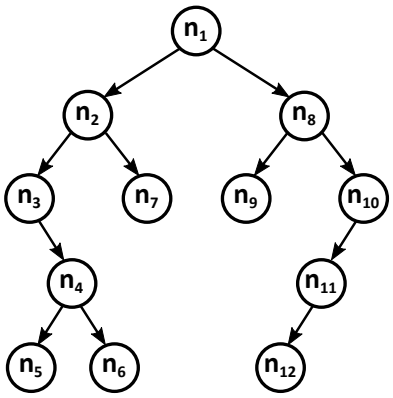
The perturbation of the existing solution is performed in two steps: (i) first, a node is selected at random and removed from the tree; then, (ii) the removed node is reinserted at a new random location in the tree. Once the tree is perturbed, its new cost is computed by converting it into a floorplan and measuring the total wirelength. However, deleting and inserting a node can affect the other nodes in the tree.

Deletion

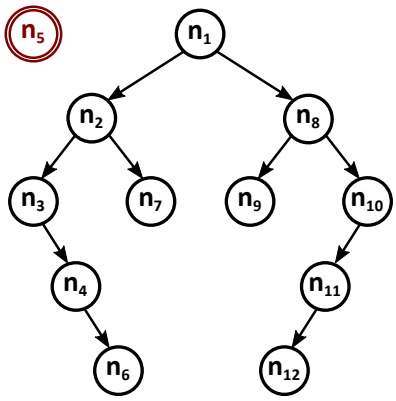
The process of deleting a node from the tree varies depending on the type of node: (1) a leaf node with no children, (2) a node with only one child, and (3) a node with both children. Figure 4.3 shows examples on how to delete the node in each of these three cases. Deleting a leaf node simply removes the node without affecting the rest of the tree. If the node has only one child, then it is removed, its child is moved up to replace it and the child's subtree is moved with it. However, in the case where the node has both children, one child is randomly chosen to replace the deleted node. Then, the subtree on that branch is rearranged, recursively, by randomly moving one child up to fill the gap.

Insertion

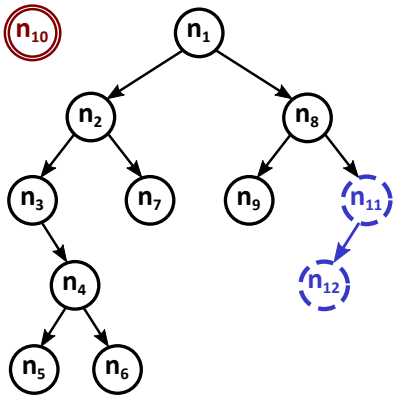
To insert a node n_j into the tree, a node n_i from the tree is randomly chosen to become its parent. One of the two branches of n_i is selected at random and n_j is added on that branch. If the parent does not already have a child on that branch, the node is directly added without any further consequences. However, if the parent n_i has already a child on that branch, then when n_j is inserted, it becomes the new parent of that child. With this approach, the inserted node will have one child at most. Figure 4.4 illustrates the insertion process with examples that cover the two different cases.



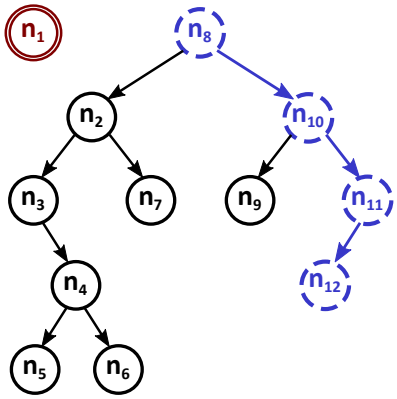
(a) Initial B*-tree.



(b) Deleting a leaf node (n_5).

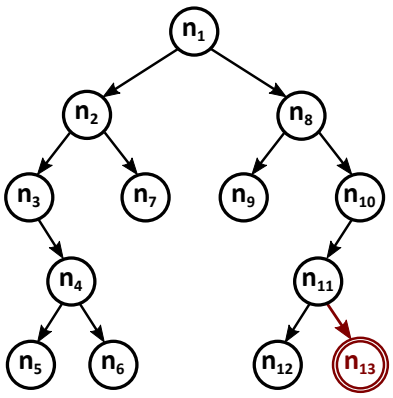


(c) Deleting a node with one child (n_{10}).

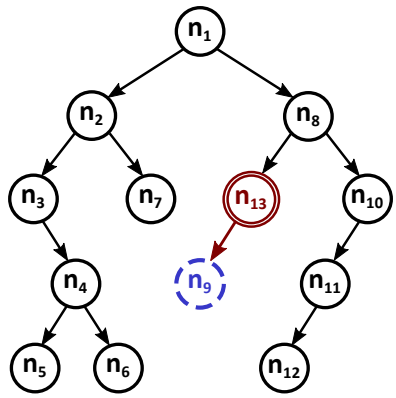


(d) Deleting a node with two children (n_1).

Figure 4.3 – Deleting a node from the B*-Tree in the cases where the node is (b) a leaf node, (c) a node with only one child, or (d) a node with two children.



(a) Inserting a new node (n_{13}) after a leaf node (n_{11}).



(b) Inserting a new node (n_{13}) after a non-leaf node (n_8).

Figure 4.4 – Inserting a node into the B*-Tree of Figure 4.1 in the cases where the parent is (a) a leaf node or (b) a node with existing children.

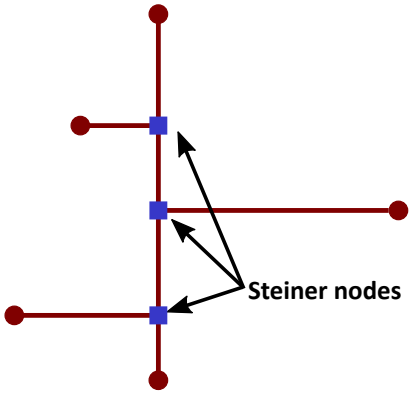


Figure 4.5 – The Rectilinear Steiner Minimal Tree (RSMT) connecting 5 nodes by adding 3 new Steiner nodes.

4.3.3 Cost Function

The goal of the floorplan is to reduce the wireload effect by minimizing the length of the different interconnects. So the total wirelength is used as a measure of the floorplan's cost, guiding the optimization algorithm in the search for an optimal solution.

One of the most common techniques used to minimize the wirelength in the floorplanning of VLSI designs is the *Rectilinear Steiner Minimal Tree (RSMT)* [Hwang et al., 1992]. It determines the tree that connects a set of nodes using the minimum wirelength, measured in Manhattan distance. To build this tree, new nodes, called *Steiner nodes*, can be added. Figure 4.5 shows the RSMT that connects 5 nodes while adding 3 Steiner nodes.

There exist several algorithms to solve the RSMT problem [Hwang et al., 1992], and one of the fastest approaches is a lookup table based RSMT algorithm known as *FLUTE* [Chu and Wong, 2008]. FLUTE relies on precomputed RSMTs for low degree nets (up to 9 nodes per net), stored in look-up tables. It then uses a net-breaking technique to convert high degree nets into sets of low degree nets whose RSMTs can be found in the look-up tables. This technique can determine the net's tree and its length very fast and accurately for any number of nodes.

We use FLUTE to compute the rectilinear Steiner minimal tree and its length for each net. Then the cost used by the simulated annealing algorithm is the sum of the lengths of all the nets in the architecture.

Having the wirelength as the only metric in the cost function can result in final clusters of irregular dimensions, if they lead to shorter wirelength. So, although not considered here, the final aspect ratio of the cluster can be added to the cost function, increasing the cost of any solution that is far from the desired aspect ratio.

4.3.4 Temperature

The temperature T is used to control the simulated annealing search, allowing for a wider search space at the beginning of the algorithm but then a convergence to an optimal (or near optimal) solution towards the end. The algorithm needs to determine first the value of the initial (high) temperature for a better control of the uphill probability.

To compute the initial temperature T_0, a random B*-Tree is perturbed and, if the new tree has a worse cost than the previous one, the difference in cost Δcost = cost(new) − cost(old), which is the uphill cost, is saved. This process is repeated over multiple randomly generate B*-Trees perturbed several times and the average uphill cost Δavg is computed. The initial temperature should be set much higher than the average uphill cost, so it is computed using

$$T_0 = \frac{\Delta avg}{\alpha}, \tag{4.2}$$

where α is a tuning parameter of the algorithm.

The temperature is gradually decreased, in a controlled way, so that worse solutions are less likely to be selected as the algorithm advances, leading to an eventual freezing of the system around the best seen solution. As shown in Algorithm 1, the new temperature is determined by

$$T = \lambda^n T, \tag{4.3}$$

where n is the number of times the temperature is decreased and λ is a variable used to control the speed with which the temperature drops.

4.4 Integration in FPRESSO's General Flow

This wire modeling approach with its floorplanning algorithm of Section 4.3 are integrated into the general flow of FPRESSO. Although the flow itself will be discussed in details in Chapter 5, we will briefly overview the way the algorithm interfaces with the remaining parts of the flow.

As previously mentioned, FPRESSO takes a description of the architecture, written in XML format, and identifies the different blocks and interconnects within the cluster. Aiming at modeling the wires first, FPRESSO starts by converting the architecture into a network of modules ready to be placed, and creating the B*-Tree in such a way that each module is associated with a node in the tree.

The width and height of each module is required before running the algorithm, to be able to correctly place the modules and abide by the geometric relationships of the B*-Tree. But since the cluster has not been modeled yet, we do not know at this stage the area of its components.

Thus, we use the average area of each component over its different sizings (i.e., the multiple instances of the cell, each with a different drive strength). This data is easily derived, since the different components with all their characteristics and sizes are stored in the library of FPRESSO. It is assumed, for simplicity, that each component has a square shape (aspect ratio of 1), although the aspect ratio is added as a parameter and can be easily modified. Doing this floorplanning during the modeling process, we have no information on the final layout of the architecture or the pin locations of the floorplan modules, so we assume that a wire starts/ends at the centroid of each module. Once the cluster is floorplanned and the placement optimized, the length of the nets is computed along with their respective capacitance and resistance. Then the FPRESSO design is annotated with this information before optimization.

Once optimized, the actual area of each architectural block is known and, as such, its correct dimensions are calculated, so the floorplan can be corrected accordingly. The cluster is then floorplanned again using, this time, the exact area derived from the previously optimized circuit. Similar to the first iteration, the best solution is determined by the algorithm and the circuit is again annotated with the wires' parasitics, for another round of optimization in FPRESSO, before returning the final results to the user. Floorplanning the architecture again, while using accurate area measures, helps minimize the noise added by the initial area assumptions. Having only two floorplanning iterations is enough to achieve the desired results, as will be shown in the experiments of Section 4.5.

This wire modeling approach relies on several assumptions like, for instance, that the wires are connected through the centroids of the components. This assumption, for example, is needed to determine the beginning/end of wires especially since the design components are only modeled and the actual layout is unknown, so there is no indication on where the pins of each component are placed. This assumption can be easily modified in the algorithm, but we believe that it does not have a major consequence since we are simply modeling the architecture (components and wires) and not providing a final layout of the design. Another assumption is that the SRAM configuration bits are spread across the different components in which they are used, while, in reality, they are grouped into columns that span the FPGA. We realize that this is probably an important aspect of the FPGA floorplanning, but we focus mainly on having a quick and automatic estimation of the intercomponent wireloads, as part of our FPGA modeling approach.

4.5 Experiments

Although we do not present the complete flow of FPRESSO till Chapter 5, we will anticipate some results relevant to the wire modeling approach, to evaluate its effect on the performance of FPRESSO, in general. The overall modeling of FPRESSO itself will be extensively evaluated in the subsequent chapters.

Once the simulated annealing algorithm is well tuned, by varying the different parameters of the heuristic (such as λ and M), the efficiency of the floorplanner is evaluated and tested it on

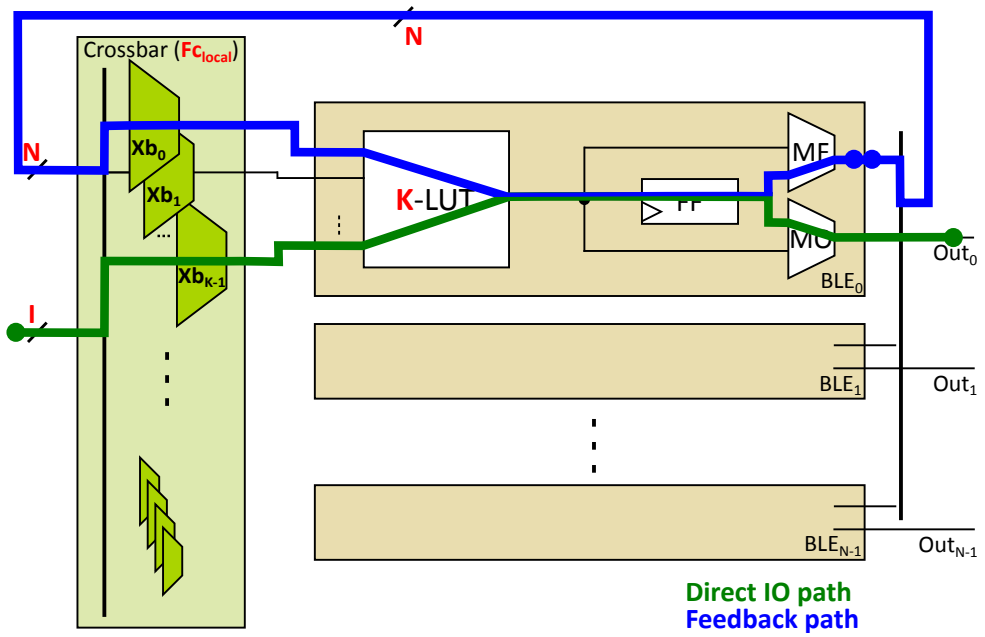


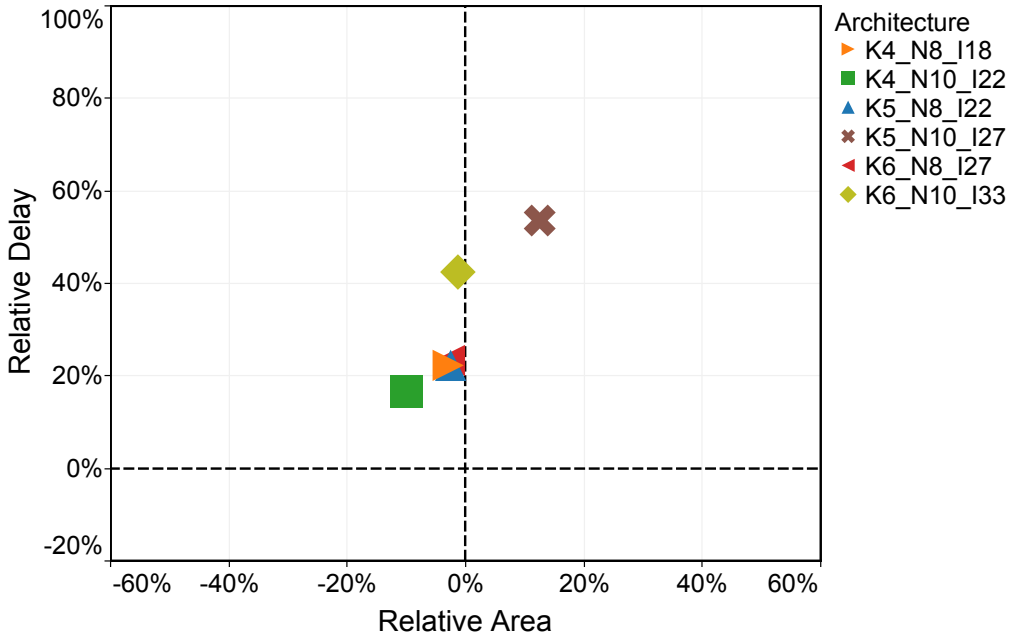
Figure 4.6 – A standard FPGA cluster with N BLEs, K-input LUTs, I cluster inputs, and an input crossbar.

a set of FPGA architectures. We split the evaluation into two sets of comparisons: (i) FPRESSO with and without wire modeling, and (ii) our wire modeling approach with the one used in COFFE.

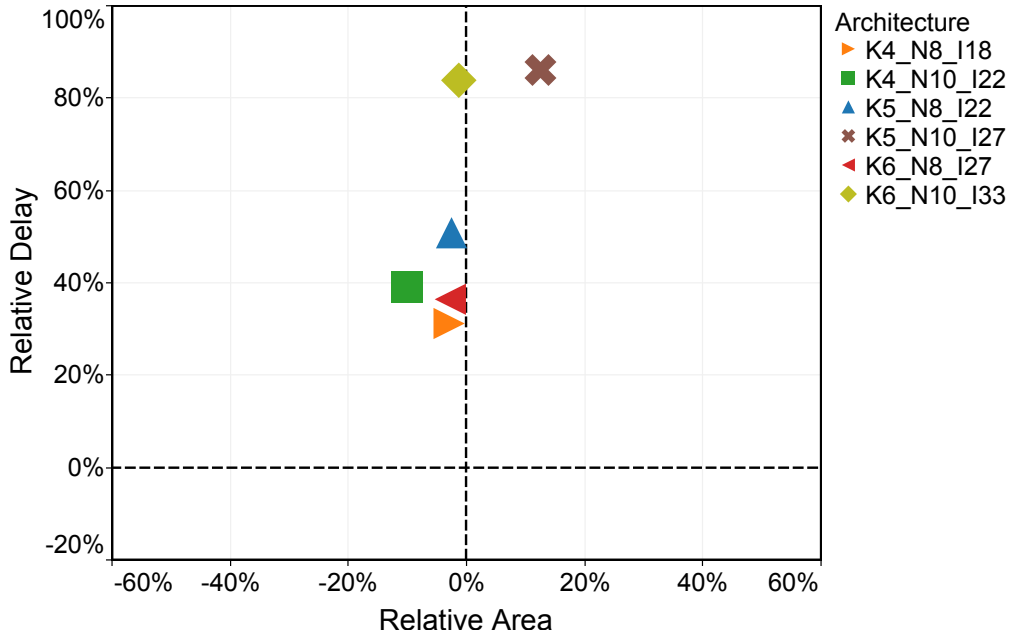
4.5.1 Comparison Within FPRESSO

To evaluate the effect of wire modeling on the results of FPRESSO, we model a set of architectures in (1) FPRESSO without wire modeling and (2) FPRESSO running with the wire modeling approach, as explained in Section 4.4. Figure 4.6 shows the general structure of the tested architectures. Similar to the Altera Stratix FPGAs, the architectures are composed of N BLEs where each BLE contains a K-LUT, a flip-flop, and two multiplexers, one for local feedback (MF) and one for global output (MO). The cluster also contains a fully populated input crossbar ($F_{c_{local}}$), designed as a set of multiplexers (Xb0, Xb1, etc.) connecting the I cluster inputs and the N local feedback signals to the inputs of the LUTs.

Figure 4.7a shows the relative delay and area of the direct input-output path, for a set of architectures, measured using FPRESSO with wire modeling, with respect to a version of FPRESSO that does not include the wire modeling iterations. The direct input-output path starts from a cluster input to a cluster output, going through the input crossbar, LUT, flip-flop, and output multiplexer, as shown in Figure 4.6. The results show that considering the wireload effect during the modeling process increases the delay by about 20% to 60%, depending on the size of the architecture and the optimization,while maintaining almost the same area. This increase in delay is mainly due to the wire delays and loads that are not accounted for without



(a) Direct input-output path



(b) Feedback path

Figure 4.7 – The effect of wire modeling on FPRESSO's results. The relative delay and area of two representative paths of an FPGA architecture, modeled in FPRESSO with wire modeling and compared to a version of FPRESSO without wire modeling.

Table 4.1 – The area correction after the second and third iterations of the floorplanning algorithm.

Architecture			Area Correction	
K	N	I	2nd Iteration	3rd Iteration
4	6	14	3.36%	0.88%
4	8	18	8.07%	4.17%
4	10	22	3.30%	1.46%
5	6	17	10.42%	3.71%
5	8	22	2.69%	0.91%
5	10	27	8.73%	6.66%
6	6	21	5.23%	3.59%
6	8	27	1.75%	1.83%
6	10	33	3.05%	3.78%
Average			5.18%	3.00%

the wire modeling algorithm.

Moreover, we observe a similar behavior in results between the direct input-output path and the feedback path which starts from the output of the flip-flop to its input, going though the feedback multiplexer, input crossbar and LUT (also shown in Figure 4.6). Despite an increase in the relative delay by about 25%, on average, for the feedback path with respect to the direct one. The feedback path is usually assumed to be a long wire connecting the output multiplexer to the input crossbar; however, our algorithm does not only measure the length of the wires but also floorplans the FPGA cluster in order to minimize the total wirelength. This means that the floorplanner tries to place connected components as closely as possible to reduce the delay of that connection. As such, the feedback multiplexer could be placed near the multiplexers of the input crossbar, reducing the feedback delay. Figure 4.8a shows the floorplan of a simplified cluster with one BLE using our floorplanning algorithm. It clearly shows how the feedback multiplexer is placed close to (1) the input crossbar multiplexers and (2) the LUT and flip-flop, reducing the feedback delay.

As explained in Section 4.4, our algorithm iterates twice: in the first iteration it uses an average area of the components to floorplan; but, once the architecture is optimized (as part of FPRESSO's general flow), we identify the actual areas of the used components and run the algorithm again with the correct measurements. Table 4.1 shows the difference between the average areas used in the initial iteration and the ones obtained after the optimization, and used in the second iteration of the algorithm. There is, on average, about 5% less total area than what was initially approximated, indicating that FPRESSO's library has slightly bigger components than what is generally used during the optimization of these architectures. However, this difference is so minimal that it can even fall within our modeling margin of error. Table 4.1 shows also the area correction if a third iteration is added to the flow. The difference in area decreases to a negligible 3%; but with such a minimal improvement, the runtime cost of a third iteration is not justified, thus we keep only two iterations of optimizations.

Although we limit our experiments to a conventional FPGA structure, this floorplanning-based modeling approach operates at the same level of architectural flexibility as FPRESSO since it applies automatically to all architectures, even unconventional ones, and is limited only to the components that FPRESSO supports. These limitations also include the carry chain support. Since FPRESSO does not currently support carry chains, they are not considered in the wire modeling approach. However, the algorithm can be easily adapted to support this kind of dedicated, fast connections, by adding weights to the routing elements, allowing the simulated annealing to prioritize the carry chains over the other standard routing connections.

4.5.2 Modeling Comparison with COFFE

There are fundamental differences between our wire modeling technique and COFFE's. These differences can be classified into two main categories: (i) the floorplanning of the cluster and (ii) the wirelength computation.

First, COFFE assumes a fixed floorplan of the cluster irrespective of the size of the components. Since COFFE optimizes only a single representative path of the architecture, its floorplan consists of a simple linear ordering of the components as generally described in architecture files and as shown in Figure 4.8b for a single BLE. However, the placement generated by our floorplanner changes depending on the architecture and the sizes of the components. For example, floorplanning the same components of Figure 4.8b results in the placement shown in Figure 4.8a, where the modules are rearranged to minimize the total wirelength. So, for example, in COFFE, the length of the wire connecting the LUT to the feedback multiplexer depends on the width of the flip-flop. However, this dependency cannot be applied to our floorplan since it varies from one architecture to the other, and, as shown in Figure 4.8a, the flip-flop might not be placed between the LUT and feedback multiplexer.

The second major difference resides in the way the length of the nets is computed. We compute the wirelength using the rectilinear Steiner minimal tree and Figure 4.8a shows the nets used to connect the different components of the tree (not showing the input/output wires to/from the cluster). COFFE, on the other hand, ignores the height of the nets, as if all the components were stacked and measures the length along the x-axis (horizontally) only, as shown in Figure 4.8b. For example, the length of the wire connecting the output of the multiplexer Xb0 to the LUT is measured by $\text{Net}_{(\text{Xbi-LUT})}$ which is equivalent to half the width of Xb0 and half that of the LUT. However, if the rectilinear Steiner minimal tree were to be used, the wire goes through Xb1 as well, making its length equivalent to twice the width of Xb0 plus half the width of the LUT. This results in a significant difference in wirelength, especially for architectures with bigger clusters. In reality, COFFE does not start/end the wires from/at the centroids of the components the way we do; however, in order to have a relatively-correct comparison and be able to highlight the fundamental differences between the two approaches, we try to filter out any other source of difference. The comparison remains true, despite the modified assumptions.

Thus, it is clear that COFFE uses a fairly simplistic floorplanning and net representation to

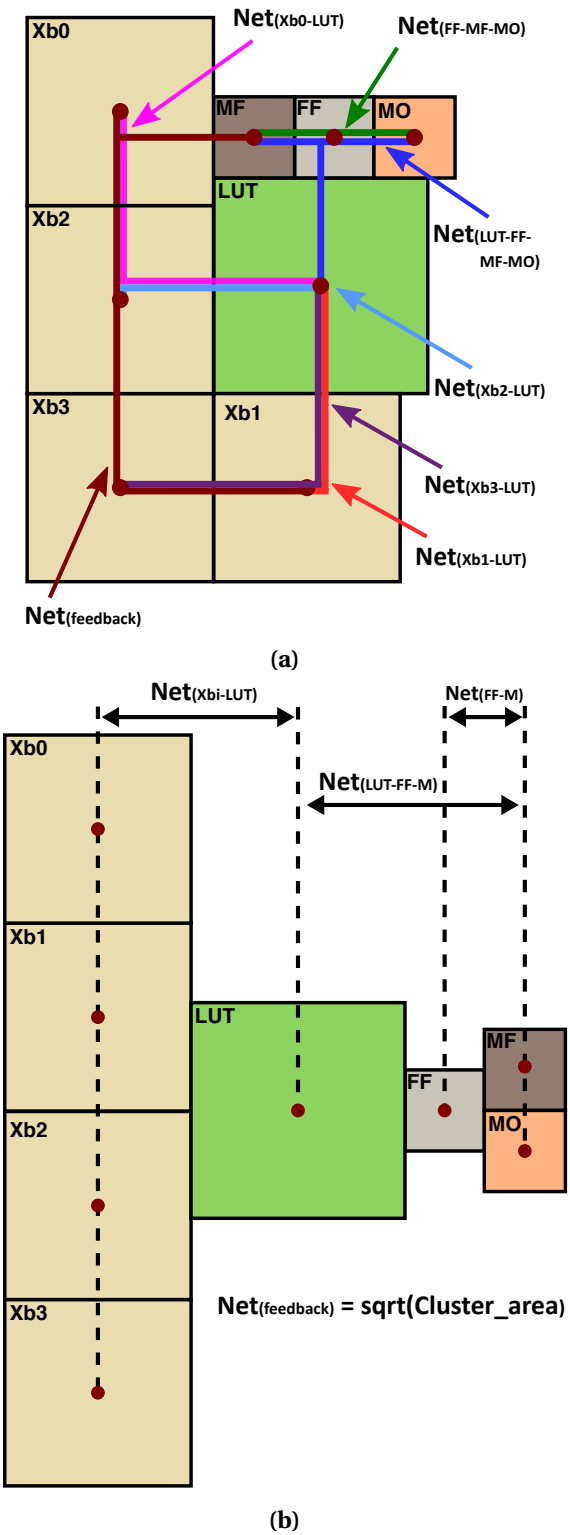


Figure 4.8 – Methodology comparison between COFFE and FPRESSO. The difference in the floorplanning and the wirelength measurement, for the architecture of Figure 4.6 but with only 1 BLE and a 4-LUT, using (a) our approach and (b) COFFE.

compute the wireloads and, by that, does not correctly account for its effect on the critical path delay. While on the other hand, our floorplanning algorithm is generic and the resulting wireload effects vary, depending on the modeled architecture.

4.6 Discussion

In this chapter, we elaborate on the details of a single, yet important, step of the general flow of our modeling approach. We present the wire modeling method used to account for the wire parasitics while modeling the FPGA architecture. It is based on a simulated annealing algorithm that floorplans the different components of the FPGA cluster while minimizing the total wirelength, before estimating the length, capacitance, and resistance of all the cluster wires. We show, in the process, that wireloads have a substantial effect on the delays of the FPGA architectures.

We do not ignore that many other factors can influence the floorplanning of FPGA components (e.g., SRAM placement), we believe that this approach introduces a more realistic methodology to assess automatically and quickly the parasitics between components. We think that this is an essential foundation to study new FPGA architectures whose clusters significantly depart from the classic multiple parallel LUT structure. We realize that with the different assumptions and approximations, this approach simply models the wires of the cluster and cannot be used directly in designing FPGAs. However, we are providing a generic method to improve wire modeling in FPRESSO, in order to have a better and fast evaluation of novel FPGAs in a vast search space of possible architectures.

Although we expanded the explanation of one part of the general flow of our modeling method, in this chapter, we will revisit the entire flow, in the next chapter, to elaborate on its various stages and key features before evaluating the approach's overall performance.

5 Architecture Optimization and Flow Automation

Perhaps one of the most important aspects of our modeling approach [Zgheib et al., 2016] is the level of abstraction it offers FPGA designers from all the modeling complexities. Being FPGA architects ourselves, who just want to quickly evaluate new ideas and architecture features, we realize the importance of a simple and fast approach that models any architecture we conceive. The technique we present is not necessarily simple, as we will see shortly; however, a complete automation of the process hidden behind a simple and convenient interface is the key to making architecture modeling seem easy for FPGA architects.

In this chapter, we focus on this automation and on the procedure followed from the moment a modeling request is made till the extraction of the modeled delay and area of the architecture. Having already built the library of cells, the general flow shown in Figure 5.1 is executed for every architecture that needs modeling.

5.1 Architecture Optimization

To obtain an area or timing model of the architecture, it is necessary to take the netlist and optimize it for some specific constraints (such as minimizing the delay along some path) in two senses: (i) every functional block can be replaced with another of equal functionality but different characteristics and (ii) buffers can be added wherever it makes sense to. Although this is only a small part of what a logic synthesizer for a semicustom flow does, it does it remarkably well and it is readily available. We thus decide to use *Synopsys Design Compiler* for the global architecture optimization phase—that is, for the optimization that needs to be run for every specific architecture a researcher is interested in exploring.

The architecture is read into the flow in a textual format that specifies the logic blocks within the FPGA cluster and their respective connections, as will be explained in detail in Section 5.3.1. This input description is then converted into a Verilog circuit, to be used for the optimization. Design Compiler is actually not used within its typical flow as a logic synthesizer but mainly as a drive-strength optimization tool. Reading in the circuit and the library of cells built in

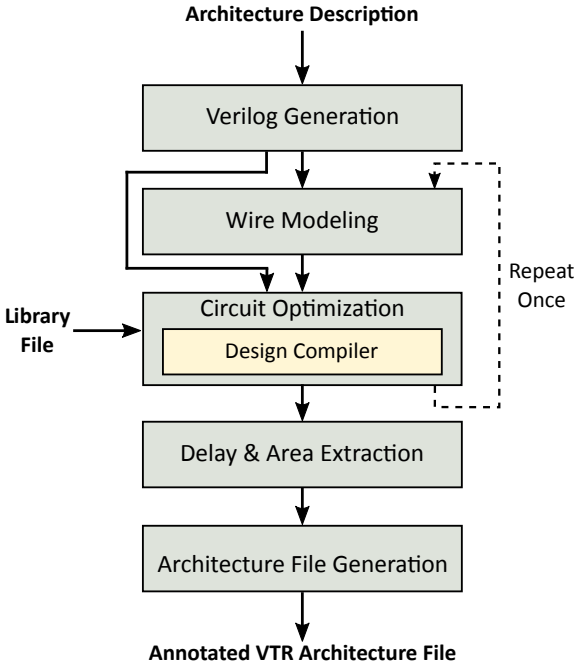


Figure 5.1 – The main flow of FPRESSO. FPGA architects provide an arbitrary description of the architecture and FPRESSO models and optimized it using the library of components, before returning a VTR-compatible architecture file complete with all the required area and timing arcs.

Chapter 3, Design Compiler optimizes the circuit by reconstructing it using cells with appropriate drive strengths and adding buffers wherever necessary. To enable this optimization-only option, we annotate each component of the circuit with the *set_size_only* attribute clearly indicating that they must be only sized and not synthesized. In order to correctly optimize the different paths of the cluster, the circuit is constrained from the inputs to the outputs, i.e. from the input of the connection block to the output of the switch block. The feedback path is also constrained from the outputs of the registers to their inputs.

The total number of buffers inserted, as well as their sizes, depend mainly on the characteristics of the architecture being optimized. Although they vary from one architecture to the other, one general trend has been observed: Design Compiler usually adds buffer on multiple and high fanout connections such as the output of a BLE that is fed back to all the multiplexers of a fully populated crossbar, for example. Design Compiler has advanced buffering techniques that automatically construct chains of inverters and determine the sizes of those chains, their optimal placements (on the different branches of the fanout tree) and the sizes of the inverters used.

The duration of this architecture optimization phase depends mainly on the number of logic and routing elements in that architecture and the number of paths that must be optimized. In general though, Design Compiler takes between tens of seconds to a few minutes, maximum, to converge to a solution, making it even more convenient for the task. Being called for every

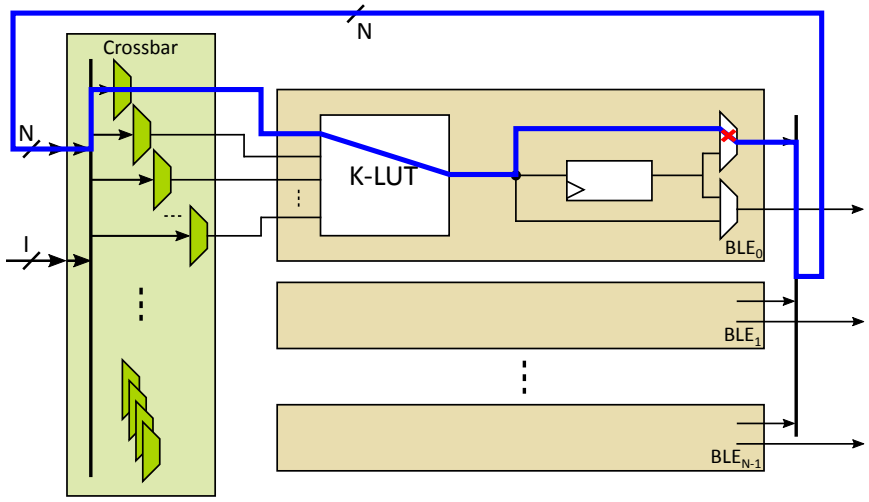


Figure 5.2 – The timing loops within the logic cluster.

architecture a researcher wants to model makes a fast optimization highly essential for an express modeling and evaluation of FPGA architectures.

However, getting a semicustom logic synthesizer to optimize a reconfigurable circuit like the FPGA architecture turns out to be a challenging task, as we will demonstrate in Section 5.2.

5.2 Optimization Challenges

The architecture optimization part of the flow seems relatively straight forward since it is entirely handled by the synthesis tool, once correctly parameterized and constrained. However, in reality, the reconfigurable nature of the FPGA architecture introduces new complications that static timing analyzers cannot anticipate. In this section, we highlight the main challenges faced when using the semicustom flow to optimize the FPGA architecture and explain the solutions we adopt to circumvent these problems.

5.2.1 Timing Loops

As a reconfigurable circuit, the FPGA cluster architecture contains, by construction, multiple combinatorial loops. This is not usually a problem since the FPGA is only used after it is configured and the configuration automatically breaks these loops. However, in our modeling flow, we consider the architecture as is, before being configured. And static timing analyzers, in general, have no notion of the reconfigurability of the FPGA, thus, they identify feedbacks as timing loops. For instance, Figure 5.2 shows one of these timing loops that starts at the feedback multiplexer and continues through the crossbar, the LUT and then back into the feedback multiplexer. This combinatorial loop does not occur in configured FPGAs, but the static timing analyzer of Design Compiler fails to identify this from the design. Luckily, it

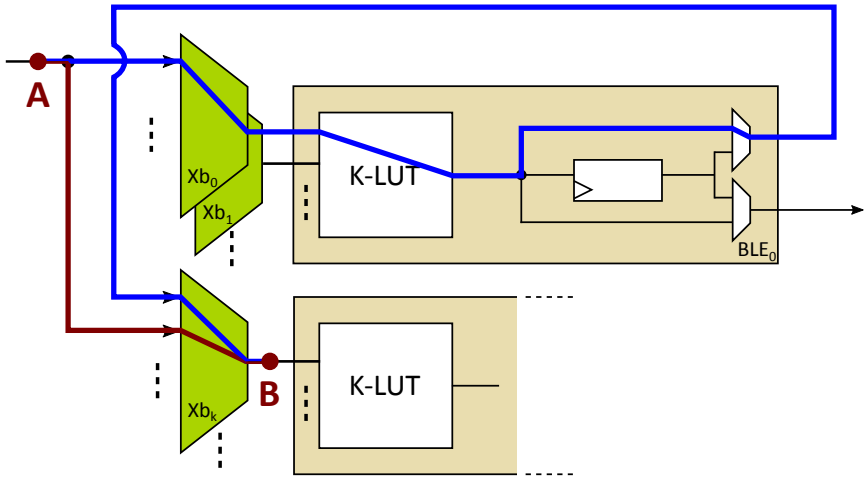


Figure 5.3 – An example of the timing paths identification problem. Measuring the delay between points A and B, Design Compiler measures by default the maximum delay by taking the long path (in blue) that goes through Xb_0, the LUT, the feedback multiplexer, and Xb_k, instead of the direct path (in red).

can signal the existence of timing loops and enumerate them. Leveraging this option, we specifically tell the optimizer that the loops do not exist, by breaking the timing paths. We do this systematically until all the loops are broken.

Even SRAMs can be seen as a loop of two inverters which is then interpreted by the static timing analyzer as a timing loop. However, since there is no interest in sizing the SRAMs, Design Compiler is instructed to ignore them (by assigning the *set_dont_touch* attribute to those inverters) so that it does not try to optimize this part of the circuit.

5.2.2 Identification of Timing Paths

In the delay and area extraction phase, we request the delay of every input-to-output path for all the components of the architecture, as will be detailed in Section 5.3.3. Although it seems to be a very straight forward task, the delays returned during the static timing analysis of our initial experiments quite often turned out to be much larger than what was to be expected, sometimes even by an order of magnitude.

When requesting the point-to-point delay between two pins, we expect to get the delay of the most direct path that connects these two points; however, static timing analysis, by default, looks for the longest possible path between the pins. Figure 5.3 shows an example of how this can occur in a typical FPGA architecture. For instance, when measuring the delay of the crossbar multiplexer Xb_k by requesting the delay between points A and B, the expected path is the one highlighted in red, that goes only through Xb_k. However, instead, Design Compiler returns the delay of the path highlighted in blue, which still connects point A to B, but through

```
<pb_type name="clb" num_pb="1" num_in="7" num_out="4">
    <pb_type name="lb" num_pb="4" num_in="3" num_out="2">
        <pb_type name="lut" num_pb= "1" num_in="3" num_out="1"/>
        <pb_type name="ff" num_pb= "1" num_in="1" num_out="1" clock="clk"/>
        <mux name="combseqfdb" input="ff.Q lut.out" output="lb.out[0]"/>
        <mux name="combseqout" input="ff.Q lut.out" output="lb.out[1]"/>
        <direct name="lutin" input="lb.in" output="lut.in"/>
        <direct name="ffin" input="lut.out" output="ff.D"/>
    </pb_type>
    <crossbar type="sparse_0.50" name="inputxbar" input="clb.in lb.out[0]" output="lb.in"/>
    <direct name="lbout" input="lb.out[1]" output="clb.out"/>
</pb_type>
```

Cluster declaration

Container declaration

Declaration of logic blocks & interconnects within the container

Figure 5.4 – An example of the user's architecture description file for the design of Figure 5.5. The architecture uses 3-input LUTs, 7 cluster inputs, 4 logic blocks per cluster, and a half-populated crossbar (i.e., $I = 7$, $N = 4$, $K = 3$, and $F_{c_{local}} = 0.5$).

crossbar multiplexer Xb_0, the LUT, feedback multiplexer, and then Xb_k.

To circumvent the problem and help the static timing analyzer identify the exact and most direct path between any two points, we need to specifically request the minimum path delay for every timing report. However, with this option, Design Compiler does not only report the shortest path but also the minimum between the rise and fall delays, which could be an underestimation of the path delay. Thus, we measure each timing path twice: requesting first the minimum rise delay then the minimum fall delay, we eventually pick the maximum of the two as the final delay of the path in question.

5.3 Automating FPRESSO

The main flow of FPRESSO uses the offline-generated library to model any user-defined architecture. With the optimization covered by a standard-cell synthesis tool, as described in Section 5.1, and the wires modeled using the floorplanning algorithm of Chapter 4, this section explains how the different parts of FPRESSO's main flow are automated.

5.3.1 Input Architecture

To abstract the modeling complexity from the user, FPRESSO takes as input a description of the architecture in XML format. The input file specifies the cluster interface, the logic components in it along with their interconnects. As shown in Figure 5.4, the architecture description is a simplified version of the one used in the VPR architecture file. In this example, the cluster has the same structure as the one shown in Figure 5.5 with 7 inputs and four 3-input LUTs (i.e., $I = 7$, $N = 4$, and $K = 3$).

In general, the architecture is described hierarchically with the help of containers, like *lb* in the example of Figure 5.4, to simplify its structure. Within the containers, the functional components, whether logic or routing elements, are declared using a set of predefined XML tags.

FPRESSO automatically identifies the structural and functional components while parsing the input file and converting the architecture into a Verilog circuit but still maintaining its hierarchy. The crossbars, for instance, are defined by the users, by specifying their inputs, outputs, and density. However, when read into FPRESSO, each crossbar is converted into a set of 2-level multiplexers with the correct inputs depending on the density and the input specification. If the crossbar is sparse, FPRESSO distributes, by default, its inputs uniformly among the created crossbar multiplexers.

FPRESSO is also designed to support fracturable LUTs which are described in the input architecture file as *modes of operation*. Each mode specifies a way in which the LUT can operate. For example, as explained in Section 2.1.2, a 6-LUT can operate as two 5-LUTs so, in the description file, two modes must be specified: (i) the 6-LUT mode and (ii) the two 5-LUT mode, along with their respective connections. FPRESSO automatically converts the description of the modes into a hardware architecture with additional reconfigurable elements, in order to enable switching between the different modes, as will be explained in details in Section 5.4.

5.3.2 Modeling Global Routing

In addition to the input description of the cluster architecture, FPRESSO needs some user-defined global routing specifications to model the switch blocks and connection blocks. Although VPR, by default, specifies dynamically the width of the global routing channel, the architecture itself has to be modeled with a specific channel width (W). Additional parameters must also be provided, such as the fraction of routing channels connected to each cluster input ($F_{c_{in}}$) and output ($F_{c_{out}}$), the density of the routing (F_s), the type of the switch block, and the wire segment length.

The switch and connection blocks are designed as 2-level multiplexers as explained in Section 3.4.3. The size of each SB/CB multiplexer is computed using the user-defined parameters, as shown in Figure 5.5. These multiplexers are added to the circuit generated from the input description and sized during the optimization phase along with the other components of the architecture. The output architecture file is annotated with the delay and sizing results of the SBs/CBs along with the routing parameters, as required by VPR.

5.3.3 General Flow and Model Extraction

One of the key features of FPRESSO is its fully automated flow that abstracts all the implementation and optimization details from the user. This flow consists of multiple steps, as shown in Figure 5.3.3, the most important of which have already been explained in details, like the wire modeling approach and the architecture optimization. The flow is automated using scripts that handle the entire process from reading in a user-defined architecture description file, until the detailed output architecture file, annotated with the modeled delay and area, is generated.

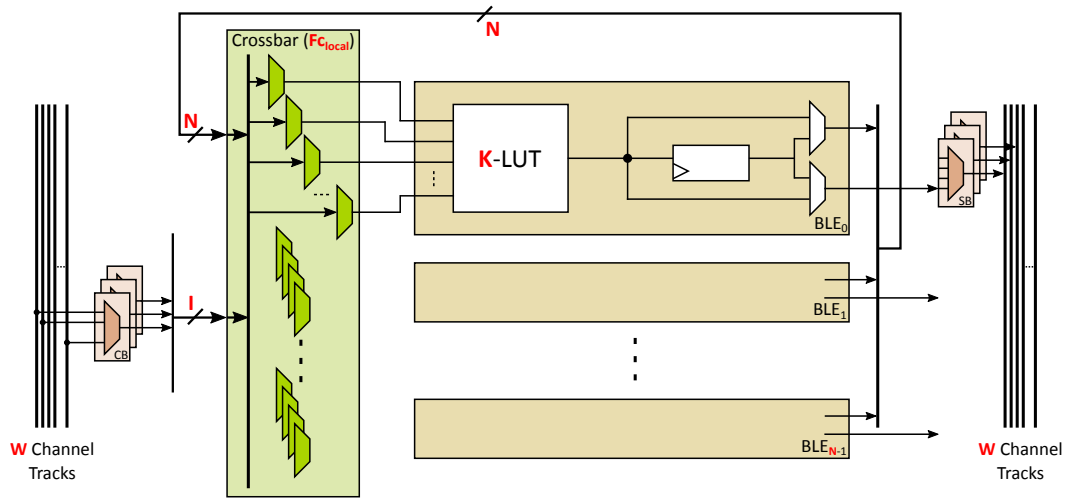


Figure 5.5 – The general structure of the FPGA tile architecture and its design parameters.

Reading in the user-specified architecture, described using the input format detailed in Section 5.3.1, FPRESSO identifies the different routing and logic elements and converts the architecture into a Verilog circuit. It then loads the circuit into the synthesis tool, Design Compiler, to locate all the timing loops and recursively break them until none remains.

Before optimizing the architecture, FPRESSO models the wires between the different components, by floorplanning the cluster, as explained in Chapter 4. Once floorplanned, the length of each wire is measured and its respective resistance and capacitance are added to Design Compiler's script as back annotation. Two rounds of floorplanning are performed: (i) starting first with an initial floorplan that estimates the component areas, the architecture is optimized in Design Compiler, and the exact area of each component is determined, so (ii) the floorplan is repeated with these new areas and the circuit is re-annotated with the parasitics, for a final round of circuit optimization.

For each of these optimizations, scripts are automatically generated to read the circuit into the synthesis tool and assign to the cells their respective attributes and to the wires their resistances and capacitances. By default, FPRESSO optimizes the architecture for delay. Starting first by setting a hard constraint of a maximum delay of zero, that cannot be met, the optimizer returns the minimum achievable delay (i.e. the critical path) so Design compiler is called for a second iteration, this time with a 10% relaxed target delay. In this iteration, the timing constraint will be met and the slack is used to reduce the optimization effect on the overall area.

Having all the needed information, the delay of every input-to-output timing arc for every logic and routing element is requested back from the static timing analysis, as well as the overall area of the cluster. Then, finally, FPRESSO generates the output architecture file, fully annotated with the delay and area, in the XML format required by the VTR flow.

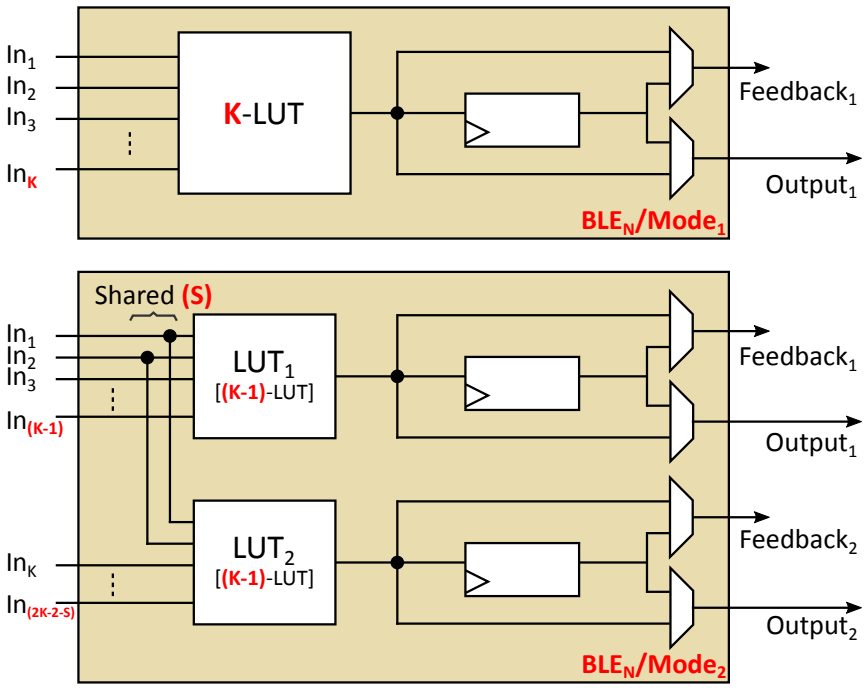


Figure 5.6 – The modes of operation of a BLE with a fracturable K-LUT.

5.4 Fracturable LUTs

There is a fundamental difference between the way CAD tools perceive fracturable LUTs and the way they are actually implemented at the hardware level. From one side, CAD tools such as VTR require a rather abstract representation of the fracturable notion, in a sense that they only require to know the *modes* in which the fracturable LUT can operate. For instance, a fracturable 6-LUT can operate either as (i) a single LUT with six inputs or (ii) two LUTs with five inputs each.

Figure 5.6 shows the two modes of operation of a BLE with fracturable K-LUT. In the first mode, $mode_1$, the BLE has a single K-LUT. The first K inputs of the BLE are directly connected to the LUT. In the second mode, $mode_2$, the fracturable LUT operates as two $(K\text{-}1)$-LUTs with S shared inputs. The BLE, in this mode, has a total of $(2K-2-S)$ inputs, distributed among the LUTs as shown in Figure 5.6.

However, from the other side, to support the two modes and have the flexibility to switch between them, the fracturable LUT has to be designed with a high degree of reconfigurability.

Since FPRESSO has to model the architecture by designing it at the hardware level while maintaining its output interface with VTR, additional reconfigurable logic for fracturable LUTs is added to translate the input modes description into a functional reconfigurable logic. This logic is then modeled and optimized, before converting it back into modes for the output architecture file. Figure 5.7 shows the hardware implementation of a fracturable K-LUT with

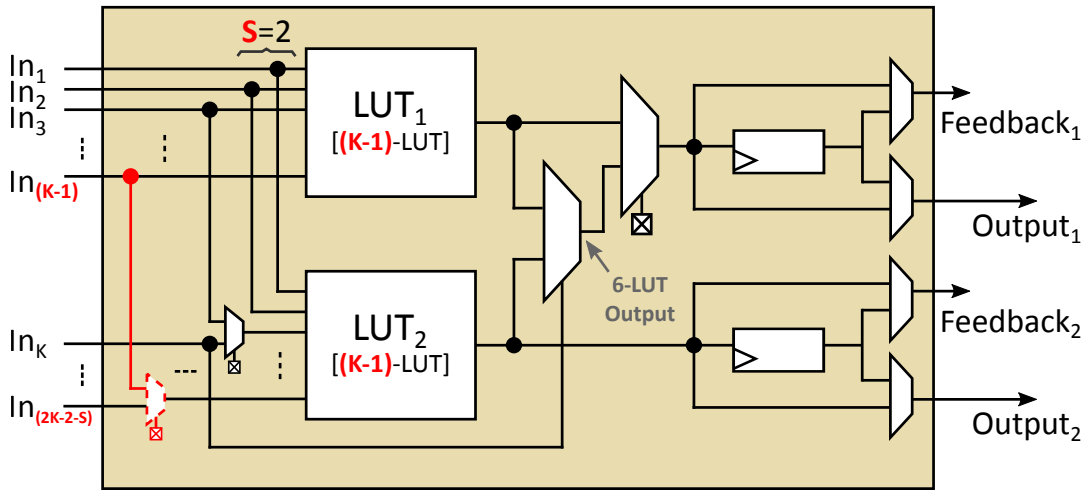


Figure 5.7 – The hardware implementation of a BLE with a fracturable K-LUT. The smallest size LUTs are used with additional output multiplexers to build the K-LUT and select the correct BLE outputs. Optional input multiplexers can also be added to ensure a correct functionality, in case of input routing constraints.

all the additional multiplexers required to support the different modes of Figure 5.6.

The implementation of a fracturable K-LUT can be divided into three phases. First, the smallest size LUT is identified and used as a logic element. In the modes of Figure 5.6, for instance, the K-LUT can be used as two (K-1)-LUTs and, as such, the smallest LUT of size K-1 is used as the basic logic element. Then, the second step consists of building the K-LUT out of the two (K-1)-LUTs by adding an output multiplexer with the K^{th} BLE input as its select. Logically, this builds a K-LUT when the inputs of the (K-1)-LUTs are identical, which will be handled in the third phase. Furthermore, in Figure 5.6, for example, the first feedback and direct BLE outputs ($feedback_1$ and $output_1$) can come from the K-LUT, if in $mode_1$, or from the first (K-1)-LUT, if in $mode_2$. So, to satisfy the output specifications of the BLE, an extra output multiplexer is added. These specifications may vary depending on the user-described architecture, so these output multiplexers and their connectivities are automatically determined from the described modes of operation.

The final phase consists of distributing the BLE inputs among the two LUTs, while enabling the two modes. Starting from the input connectivity of $mode_2$, the two LUTs share S inputs while the remaining BLE inputs are distributed among them. However, as mentioned earlier, for the two LUTs to operate as a single K-LUT, they must have the same K-1 inputs while the K^{th} BLE input is the select of the multiplexer. Ideally, the routing, which is handled by the crossbar in this case, should be able to assign the same signals to the non-shared inputs of the LUTs. However, doing that leaves no BLE inputs to connect the select of the multiplexer. Thus, an input multiplexer is needed to disconnect the K^{th} input of the BLE from the LUT inputs, by providing the option of an additional hard-wired input sharing between the two LUTs. Similarly, optional input multiplexers can also be added in the cases where the input

routing (again crossbar in this case) is not flexible enough to guarantee connecting the same signals to the inputs of the two LUTs when in $mode_2$.

This process is generalized to support any architectural constraints specified by the user, including various input/output connectivities and even nested modes, where, for example, the (K-1)-LUTs of Figure 5.7 could be fracturable as well.

5.5 FPRESSO's Performance

FPRESSO can read in an architecture description file to understand the FPGA cluster design it is modeling, and can thus be used on any architecture topology, as explained earlier. However, in order to benchmark its performance, we restrict our experimental setup to the architectures supported by COFFE, the state-of-the-art modeling tool for FPGAs.

COFFE's architectural exploration is limited to a given FPGA topology customizable through some parameters. Figure 5.5 shows the cluster architecture supported by COFFE and used in our experiments, along with its three main parameters K, N, and I which represent the number of LUT inputs, total number of LUTs (or BLEs) in a cluster, and number of cluster inputs, respectively. Accordingly, we generate different FPGA clusters by varying K, N, and I and optimize the corresponding architecture using the two modeling approaches: FPRESSO and COFFE. To limit the variance in the comparison, all local register feedbacks are disabled in COFFE (and not included in FPRESSO), the crossbar is fully populated and the LUTs are not fracturable. A 65nm UMC technology is used in a typical corner in both flows.

FPRESSO is designed to optimize the circuit for delay, by default. So, for a correct comparison, we optimize for delay as well in COFFE by doubling the delay-to-area ratio of the cost function (i.e., setting $d = 2$ and $a = 1$).

5.5.1 Runtime

One of the main objectives of our modeling approach is to enable express architecture explorations, especially over a wide search space. FPRESSO can model an architecture within minutes, although its runtime depends mainly on the size of the architecture it is modeling. To evaluate the modeling's runtime efficiency, we compare FPRESSO's runtime to that of COFFE over a large set of architectures, as shown in Table 5.1. On average, FPRESSO is about 200 times faster than COFFE. This is a game-changing difference that can enable far more comprehensive architectural explorations, as will be shown in Chapter 6.

Naturally, the observed speedup varies with the modeled architecture. However, one can notice a general trend in the variations: the speedup decreases as K, N, I, or a combination of those parameters increases, generally indicating a relative increase in FPRESSO's runtime. To further understand this behavior, we measure the runtime distribution of every step in FPRESSO's general flow for a small architecture ($K = 4$, $N = 6$, $I = 14$) and a relatively larger one

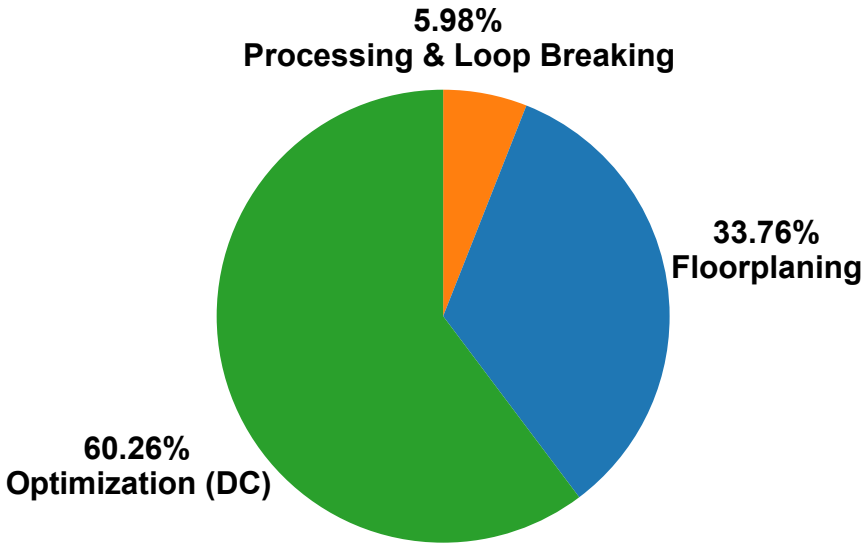
Table 5.1 – FPRESSO's runtime improvement, when compared to COFFE, for a range of architectures.

Architecture			Speedup over COFFE
#LUT inputs (K)	Cluster size (N)	#Cluster inputs (I)	
4	6	19	274
4	6	30	180
4	6	43	162
5	6	19	313
5	6	30	279
5	6	43	239
5	8	28	206
5	8	41	146
5	10	26	124
5	10	39	121
6	6	19	263
6	6	30	266
6	6	43	177
6	8	28	166
6	8	41	110
6	10	39	92
Average			195

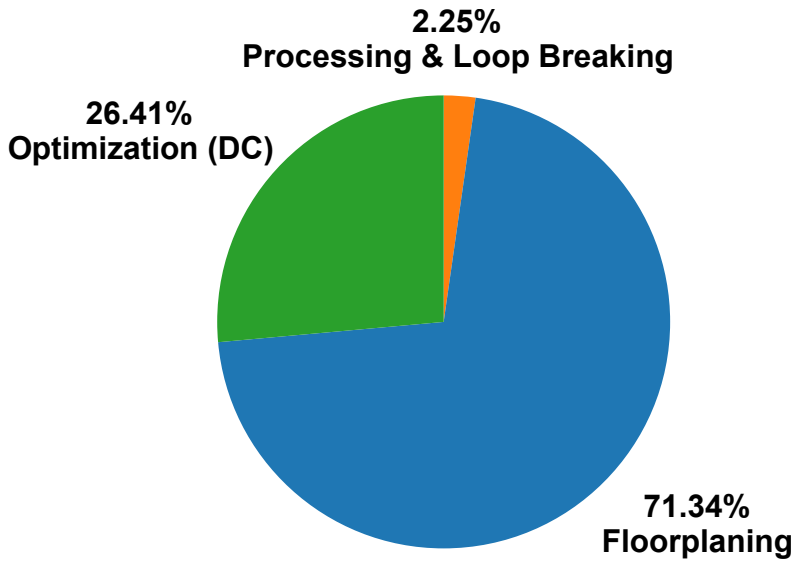
($K = 6$, $N = 10$, $I = 33$). The results, shown in Figure 5.8, indicate that, for smaller architectures, most of the runtime is spent in the architecture optimization step; however, as the architecture size increases, the runtime gets dominated by the wire modeling phase. That is mainly due to the floorplanning algorithm used to model the wires. As the architecture parameters K, N, and I increase, the number of modules to floorplan increases and, more importantly, the number and length of the wires increase, slowing down the wire modeling process. Nevertheless, despite this major shift in runtime distribution, the overall approach remains highly efficient and orders of magnitude faster than the existing tools.

5.5.2 Modeling Accuracy

To evaluate the accuracy of our modeling, we compare the delay and area estimated using both COFFE and FPRESSO, over multiple architectures, When measuring the delay, we differentiate between two delay paths: (i) the feedback path and (ii) the direct input-to-output path (called *direct IO path*, in short). Using Figure 5.5 as reference, the feedback path starts at the output of the flip-flop and goes through the feedback multiplexer, the crossbar and the LUT, back to the input of the flip-flop. The direct IO path starts from the cluster inputs and passes through the crossbar, the LUT and the output multiplexer, into the cluster output. Figures 5.9a and 5.9b show the relative delay and area measured in FPRESSO with respect to COFFE for the two paths, respectively.

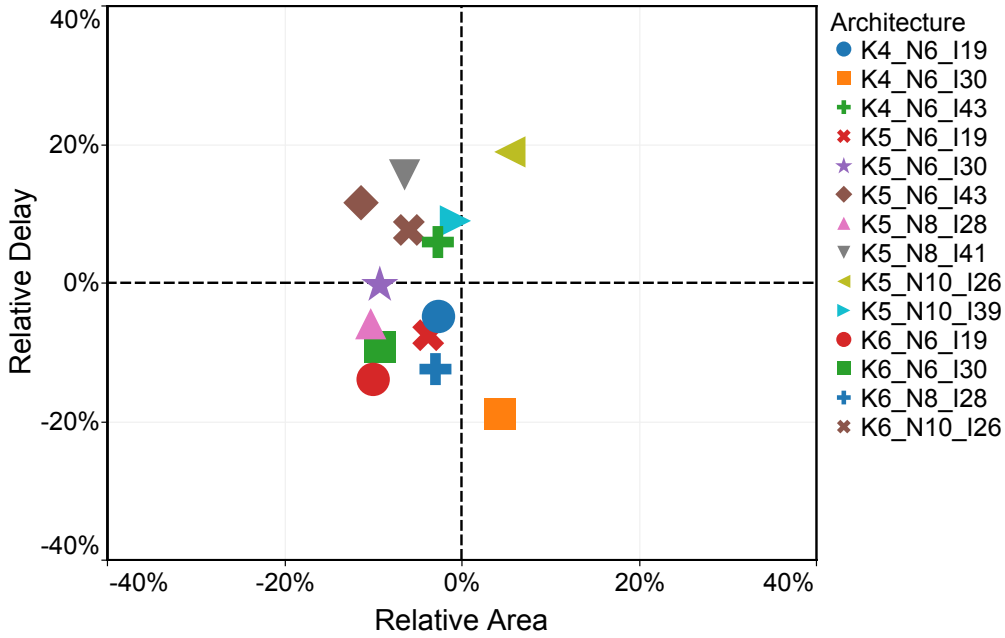


(a) Relatively small architecture ($K = 4$, $N = 6$, $I = 14$).

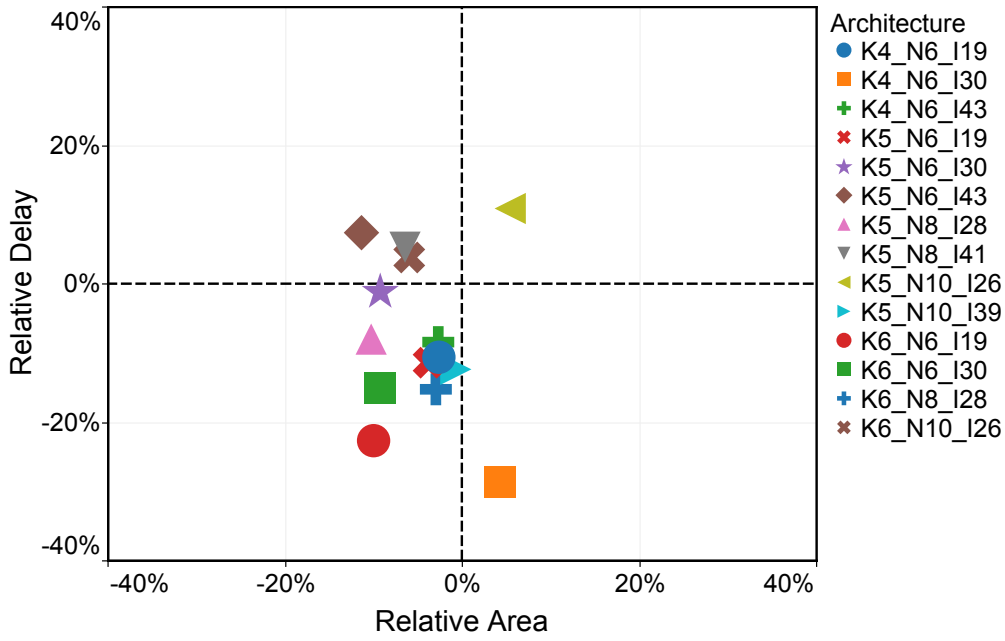


(b) Relatively large architecture ($K = 6$, $N = 10$, $I = 33$).

Figure 5.8 – Runtime distribution for two architectures with different sizes. As the architecture size increases, the floorplanning algorithm becomes the most time imposing step of the flow.



(a) Feedback path: from flip-flop to flip-flop.



(b) Direct IO path: from cluster inputs to cluster outputs.

Figure 5.9 – The delay and area of the main paths of an FPGA, modeled in FPRESSO with respect to COFFE, for multiple architectural parameters. Figure 5.9a shows the results for the feedback path, which goes from the feedback multiplexer through the crossbar and the LUT. Figure 5.9b shows the same results but for the path going from the cluster inputs through the crossbar, LUT, and output multiplexer.

We observe that the differences between COFFE and FPRESSO tend to be up to 10% in area and less than 30% in delay—mostly though in the 20% range. In general, these results are quite encouraging, since these differences account for the modeling margin of error, the fundamental differences in the wire modeling between COFFE and FPRESSO (as explained in Chapter 4), and differences in the optimization procedure—Design Compiler optimizes for a cost function that is certainly different than the one used in COFFE. Our results are mostly in the third quadrant, and slightly Pareto dominate the results of COFFE. There is a concrete possibility that our designs are marginally superior, since they benefit from the advanced optimization and buffering capabilities of Design Compiler. It is very difficult though to conceive experiments that help separate the contribution of the various differences between the two methodologies but the results are sufficiently close to be deemed satisfactory.

5.5.3 Delay and Area Tradeoff

In a different experiment, we select a single architecture (i.e., $K = 5$, $N = 6$, and $I = 30$) and try to trade off delay for area, and vice versa: For COFFE, the different data points are obtained by re-optimizing the architecture, each time using new area and delay weights in the cost function (Equation 3.1). For FPRESSO, we obtain the tradeoff by gradually relaxing the constraints in the architecture optimization phase. Figure 5.10 shows the resulting Area-Delay Pareto front of the architecture, modeled in both COFFE and FPRESSO.

Clearly, both modeling approaches offer a meaningful set of solutions. None of the two approaches is consistently Pareto dominant: COFFE performs better on area constrained optimizations, while FPRESSO can generate faster circuits. On one hand, the quality of the circuits modeled with FPRESSO depends on how rich and comprehensive the library of cells is. We suspect that FPRESSO might not be able to find more area-favored models due to the limited availability of area-optimized cells in the library (which could be the result of library pruning). On another hand, FPRESSO leverages the advanced buffering capabilities of Design Compiler allowing it to perform better signal buffering than COFFE. Nevertheless, the results seem encouraging since they indicate that both methodologies possess the ability of exploring, nontrivially, the design space. And, despite the substantial differences between the two methodologies, the results seem sufficiently consistent and reliable.

5.6 Modeling or Designing

It would be tempting to believe that our tool *designs* optimized transistor-level architectures, instead of simply *modeling* them. Although this is a tempting claim, we do not think it is a granted one. The reason is that standard cells and a classic semicustom design flow have a number of built-in electrical safeguards to guarantee functionality under any constraint; our flow, purposely, does not. For instance, standard cells never expose pass transistors to the external pins of the cell and this is one of the reasons why standard cell designs cannot match in many practical cases perfectly crafted manual designs. In our case, although on

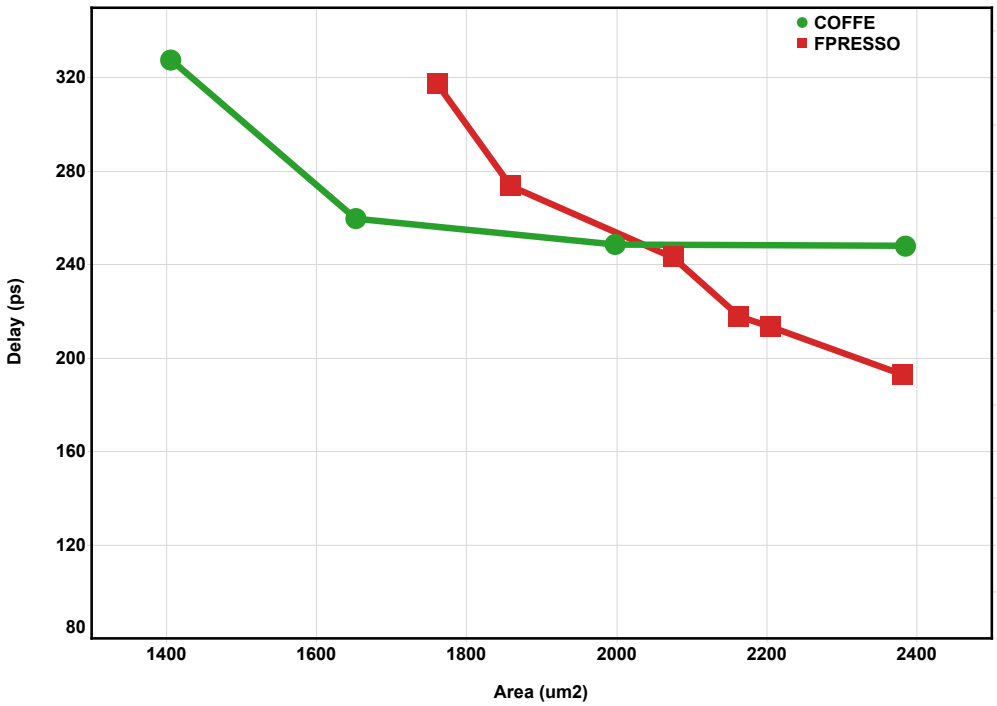


Figure 5.10 – The delay and area Pareto fronts of multiple optimizations performed by FPRESSO and COFFE, for a single architecture ($K = 5$, $N = 6$, and $I = 30$).

the outputs we always have buffers, we omit input buffers to mimic the way a hand-crafted transistor-level circuit would be built. We have studied some of the circuits resulting from our flow and, within the range of fairly conventional architectures reported here, we have not observed any electrical error; yet, our methodology is such that we do not guarantee functionality for every possible conceivable circuit. At best, we can affirm that our flow helps fast and sound modeling (our prime goal) and actively suggests architectural solutions in the buffering structure which designers may want to study in case they want to produce a production transistor-level implementation. One should note that FPRESSO benefits from the advanced buffer optimization strategies of Design Compiler which largely exceed the resizing capabilities of COFFE, for instance: Design Compiler cannot only build multistage optimal buffers when required, but can also add buffers after fanout points when the load is divided unevenly across different circuit branches.

5.7 Discussion

With this chapter, we complete the explanation of the different aspects of our FPGA architecture modeling approach. The process is entirely automated and operates at the same level of generality of VTR, the state-of-the-art academic FPGA CAD flow. It can be used to model FPGA architectures quickly, with minimum effort, and without any prerequisite knowledge in transistor-level design.

Having such an automatic architecture modeling technique that can be easily integrated into the FPGA CAD flow makes architecture explorations quite simple and almost effortless, even on a very large search space. Thus, it seems only natural that we dedicate the next chapter to architectural explorations using our automatic modeling flow. With such a fast and easy modeling method we evaluate over a thousand architectures with tens of thousands of benchmark simulations.

6 Architecture Exploration Using the Automated Modeling Technique

Perhaps the best way to evaluate our architecture modeling approach would be in its ability to reach the exact purpose it was designed for: enabling fast and accurate architecture explorations.

The latest study on FPGA architectures dates from thirteen years ago when Ahmed and Rose [2004] searched for the optimal LUT and cluster size. Having to model each architecture manually, the search space was limited to about 60 architectures. We have already established, in Chapter 5, that we are able to model an FPGA architecture within minutes, i.e., about 200× faster, on average, than the state-of-the-art architecture modeling tool. So, the 60 architectures can be modeled within hours enabling a much wider search space that can cover bigger ranges of architectural parameters or even the evaluation of the effect of other FPGA features.

Thus, to validate the correctness of our architecture modeling, we first repeat the Ahmed and Rose study [2004] by limiting our exploration to the almost-exact architectures explored in 2004 and then compare our results and conclusions to the ones reported in the study. Then, we gradually extend our exploration to a much larger search space, and focus on evaluating the effect of the crossbar density and fracturable LUTs on the FPGA [Zgheib and Ienne, 2017].

6.1 Architecture Modeling

One of the main challenges in architecture explorations is to have correct delay and area modeling of the targeted architectures. Any minor modification to the cluster requires redesigning it at the transistor level, sizing the different transistors and running SPICE simulations to measure the pin-to-pin delays of every element of the cluster. In prior explorations [Ahmed and Rose, 2000, 2004], this modeling had to be done manually, which imposed severe constraints on the feasible search space. We will show how these explorations and studies became simple, perhaps almost trivial even, with FPRESSO. We will use this section to define the explored cluster and its parameters and then overview the modeling process used in our exploration.

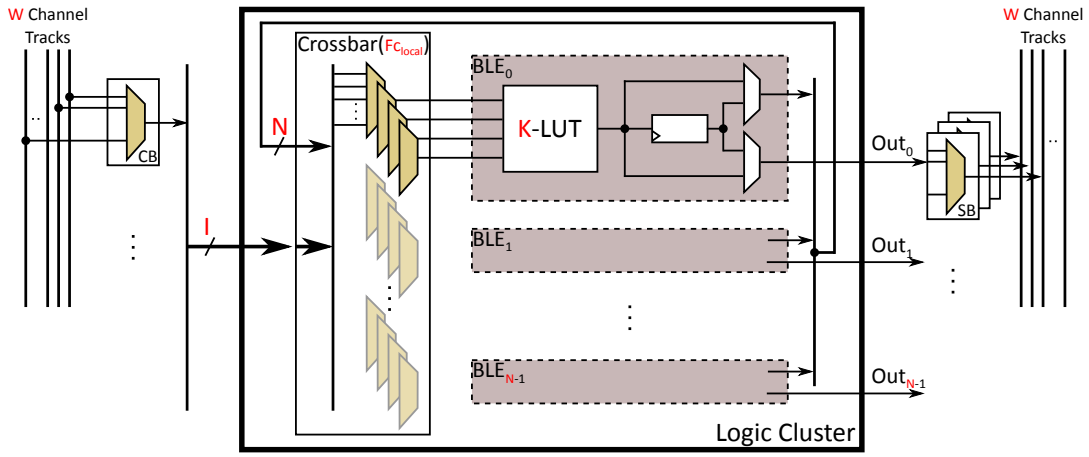


Figure 6.1 – The general structure of the FPGA tile architecture and its design parameters (K, N, etc.).

6.1.1 Cluster Architecture and Parameters

The general FPGA structure and the tile architecture have already been introduced in Section 2.1.1, however, we will highlight its key features, essential to our exploration.

We base our exploration on an FPGA architecture with the general cluster structure shown in Figure 6.1, and its different architectural parameters. Each cluster consists of N *Basic Logic Elements (BLEs)*, and has I inputs and N outputs. Each BLE has a K-input LUT, a register and two multiplexers to select between the registered and non-registered LUT output, before sending it either to the cluster output or as a local feedback. The I inputs, along with the N feedback signals, feed the input crossbar which then distributes them to the BLEs (and hence the LUTs). As such, the crossbar has $(I+N)$ inputs, $(N \times K)$ outputs and a density $F_{c_{local}}$ which indicates the fraction of the inputs connected to each output.

The inputs and outputs of the cluster are connected to the global routing through Connection Blocks (CBs) and Switch Blocks (SBs). The fraction of routing channels connected to each of the cluster's input/output is defined by the parameters $F_{c_{in}}$ and $F_{c_{out}}$, respectively.

6.1.2 General Modeling

We use our automatic modeling approach, FPRESSO[1], to model the explored architectures. As detailed in Chapter 5, it automatically models the FPGA by taking as input a description of the cluster architecture, in a simplified XML format and returns the modeled architecture, fully annotated with the area and delay of every element. The output file generated by FPRESSO follows the exact XML format requirements of the packer of VTR, which makes it highly convenient and allows it to be easily integrated in the CAD flow.

[1] Available online at fpresso.epfl.ch

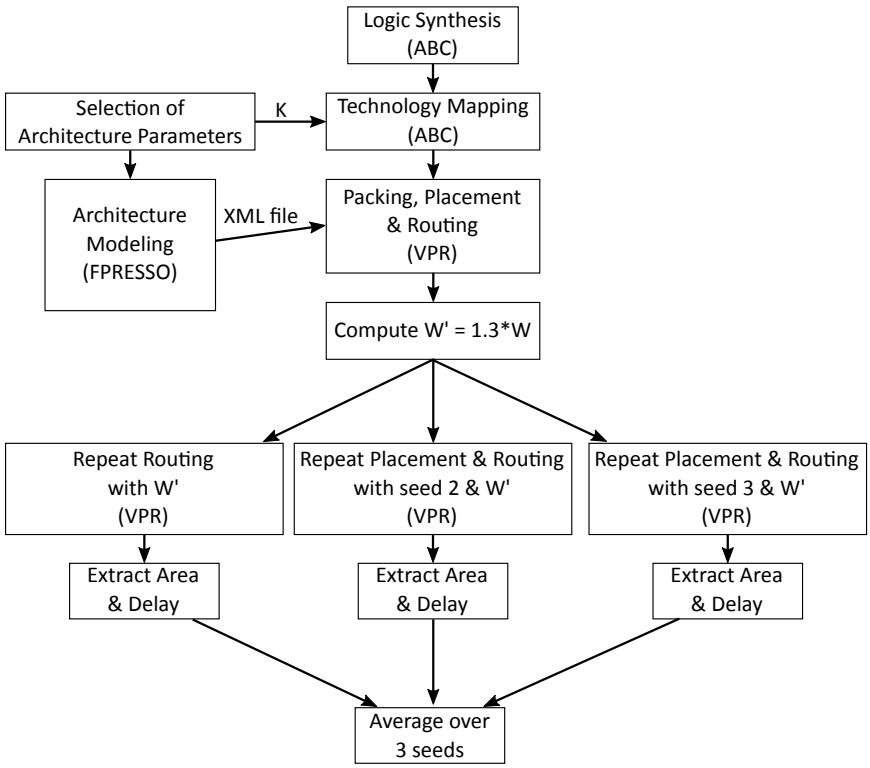


Figure 6.2 – The different steps of the experimental methodology.

Given that the prior work does not specify the routing density, we choose the default values of $F_{c_{in}}$ and $F_{c_{out}}$ (0.15 and 0.10 respectively) used in the architecture files provided with the VTR flow, with a Wilton switch box and a segment length of 4. We model the architectures using FPRESSO in a 65 nm UMC technology (typical corner).

To model global routing, Ahmed and Rose [2004] assume that the routing buffers scale proportionally with the length of the tile, while basing it on a certain assumed size for the small architecture of $K = N = 4$. To maintain consistency and properly compare with that study, we scale the routing buffers the same way; however, instead of assuming the size of the base architecture, we model it along with all the routing multiplexers and use the sizes reported by FPRESSO.

6.2 Experimental Methodology

To facilitate the exploration of a very wide search space, the entire experimental process is fully automated. In this study, we model and explore over 1,200 architectures, as opposed to the 60 architectures tested by Ahmed and Rose [2004], and run more than 41,000 benchmark simulations. Such an extensive exploration could not have been possible if the experimental setup was not fully automated.

6.2.1 General Flow

The experimental flow consists mainly of two phases: (1) architecture modeling and generation of architecture description files, and (2) running all selected benchmarks on each modeled architecture. The different steps of these phases are shown in Figure 6.2.

In the first phase, architectures are automatically modeled using FPRESSO: We only have to create a template of the desired architecture and vary the cluster parameters shown in Figure 6.1 before sending to FPRESSO a description of the cluster-to-be-modeled, using its input architecture format. The resulting output is an architecture description file, in XML format, as required by the packer.

The second phase handles the benchmark simulations on the modeled architectures. Each pre-elaborated benchmark is first synthesized and technology mapped, knowing the size of the LUT (K) in the architecture, using the synthesis and verification tool ABC [ABC]. In all our experiments, we use the default ABC version that comes with VTR 7.0 [Luu et al., 2014a]. Having FPRESSO generate the architecture description file, the benchmark is packed, placed, and routed using VTR 7.0, with unlimited routing constraints. Then, knowing the minimum channel width (W) required to route the benchmark, the channel width is increased by 30% and the routing step is repeated but now with a fixed channel width ($W' = 1.3 \times W$). Placement and routing is also repeated, for three different placement seeds, and the extracted delay and area results are averaged (over these seeds), to filter out the placement noise.

6.2.2 Benchmark Selection

There seems to be a growing consensus in the FPGA research that the MCNC benchmarks [Yang, 1991] are a rather outdated benchmark suite that does not represent realistic circuits on which the FPGA might be used. Even the big 20 MCNC circuits are often criticized for being small with some purely combinatorial designs and no heterogeneous circuits (memory and DSP blocks) [Murray et al., 2013]. This encouraged introducing new benchmark suites, such as the VTR benchmarks, in the VTR project [Rose et al., 2012], and the Titan benchmarks [Murray et al., 2013], as better alternatives to the MCNC benchmarks.

We set out to verify whether the MCNC circuits can be relied on in such architectural explorations. So, we design an experiment to compare the MCNC and VTR benchmarks over the same set of architectures, modeled in FPRESSO so that the exact cluster delays and areas are used in both cases, even if the VTR benchmarks require additional RAM and multiplier blocks. Since FPRESSO does not model RAM and DSP blocks, we use the delays and areas provided in the VTR 7.0 architectures, scaled to the correct technology node. We test the benchmarks of each suite on 60 different architectures with a large range of K and N, and a fully populated crossbar.

Figure 6.3 shows a comparison between the delay-area products obtained for each of the benchmark suites. Clearly, the scatter follows a linear trend indicating that the conclusions

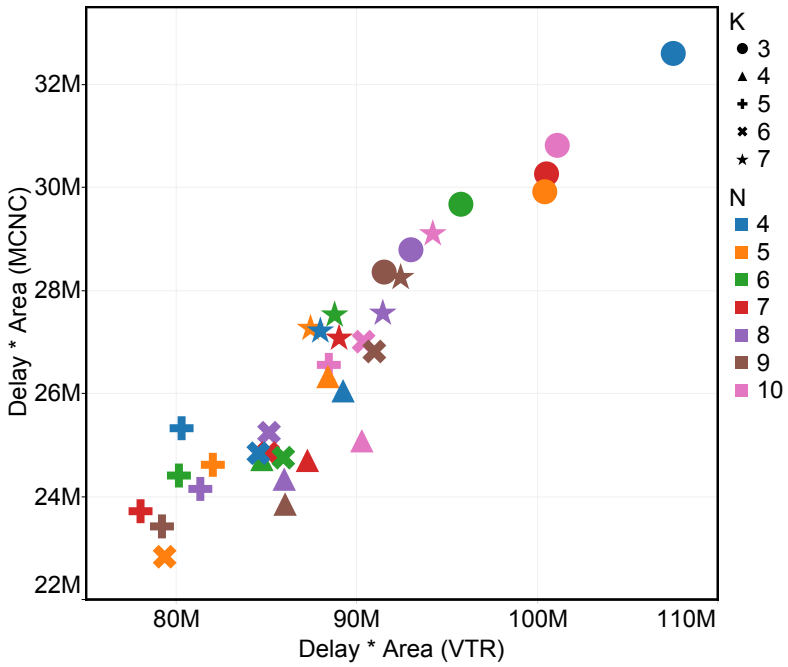
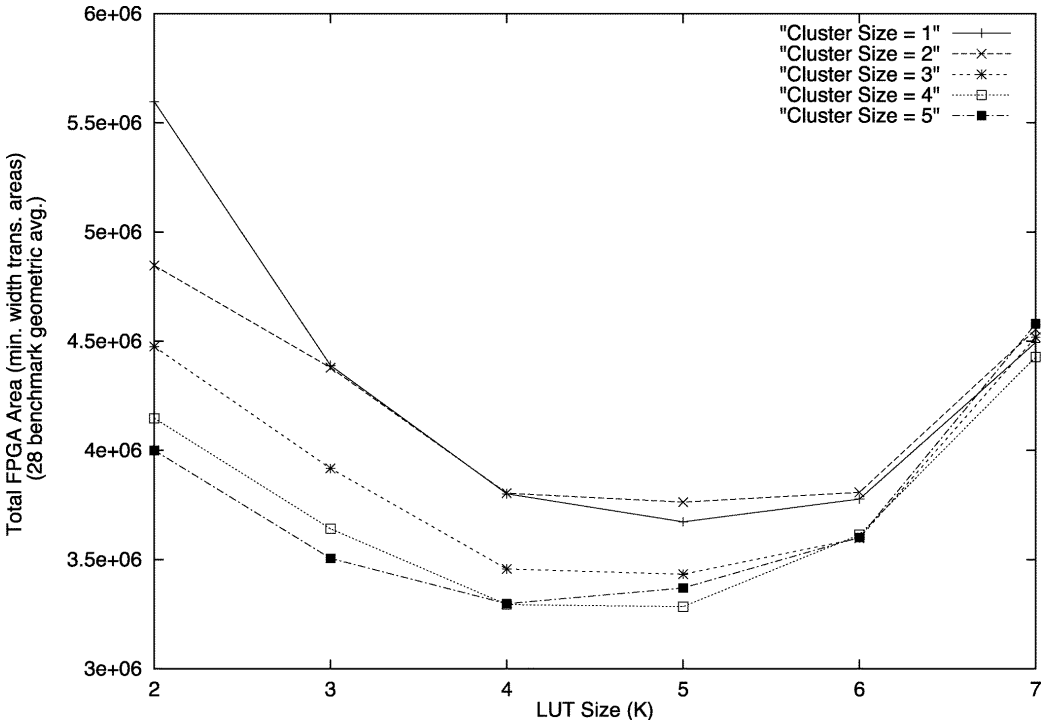


Figure 6.3 – Comparison of the delay-area product for the MCNC and VTR benchmark suites, over multiple K and N. The results of the two benchmark suites have a very similar behavior.

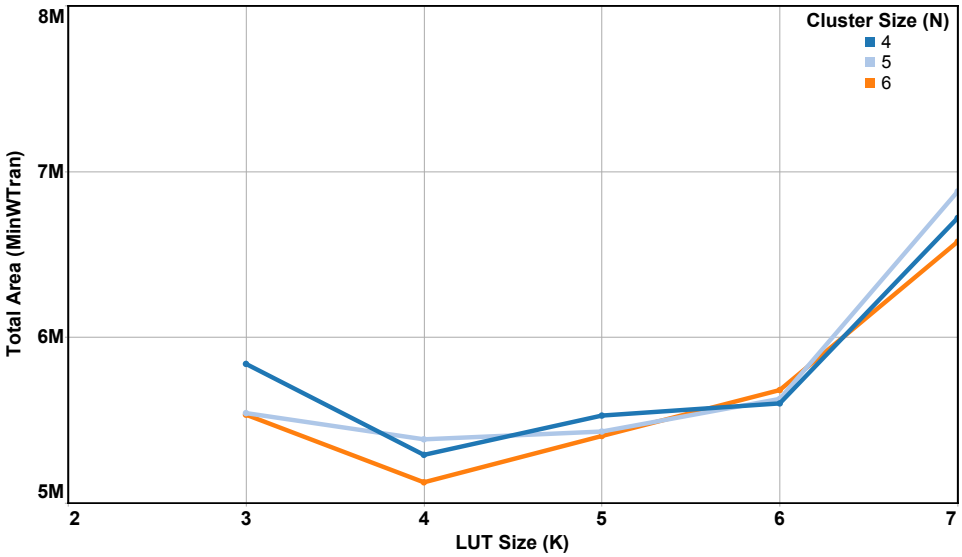
derived in such architectural explorations are valid using either of the two benchmark suites. There is no denying that the VTR benchmarks are bigger and have longer critical paths, resulting in about 3× more delay-area product. However, this increase in size barely causes any deviation in the results since this ratio is maintained with very minor variations over the large set of architectures. This shows that the MCNC benchmarks can still be representative of the performance of an FPGA architecture and we would even claim that they are more advantageous in architectural explorations since they can lead to similar conclusions for a shorter experimental runtime. Hence, we decide to use the MCNC benchmarks in our experiments. We realize though that, if carry chains were to be added to the architecture, these conclusions might need to be re-evaluated since the VTR benchmarks might benefit from the hard adders and fast chains more than the MCNC benchmarks.

6.3 Revisiting Existing Studies

We start first by re-evaluating the latest studies on the effect of the different cluster parameters onto the FPGA performance. So, we limit our initial experiments to the search space of the these studies, with some minor differences. There are, in fact, several unspecified variables in the Ahmed and Rose papers [2000; 2004], like the fraction of routing channels connected to each cluster input/output, $F_{c_{in}}$ and $F_{c_{out}}$, for example. As mentioned in Section 6.2.1, we use the default $F_{c_{in}}$ and $F_{c_{out}}$ values given in the VTR architecture files, and this forces us to limit the cluster size to a minimum of 4, otherwise some cluster inputs would not be able to



(a) Total area with respect to the LUT size (K), for small cluster sizes, from the reference study [Ahmed and Rose, 2004].



(b) Total area with respect to the LUT size (K), for small clusters of sizes (N) 4 to 6, measured using our methodology.

Figure 6.4 – Total area with respect to the LUT size (K), for small cluster sizes, as measured in both the reference study and our experiments.

reach any routing channel. Additional limitations are introduced by the CAD tools where, for example, ABC does not map on only 2-LUTs.

Taking all these constraints into consideration, the range of the exploration space is determined by the following parameters:

- $3 \leq K \leq 7$,
- $4 \leq N \leq 10$,
- $I = \frac{K}{2} \times (N+1)$, as used in the reference study [Ahmed and Rose, 2004], and
- $F_{c_{local}} = 1$ (i.e., a fully populated crossbar).

We test these architectures on the big 20 MCNC benchmarks and represent the results, reported using the geometric mean over all benchmarks, in the same format as in the original study.

Figures 6.4 and 6.5 show the total area, measured in minimum width transistors (MinWTran) as the LUT size varies from 3 to 7, for cluster sizes between 4 and 10. The results from Ahmed and Rose's study [2004], for the same parameters are also added for comparison. Figure 6.6 shows the total delay for the same architectures and compares the reference results to ours. When compared with the prior work, one can clearly see the same behavior of the area and delay curves as K and N vary. Certainly, the range of the area/delay values changes due to the differences in technology; however, the same trends are generally preserved.

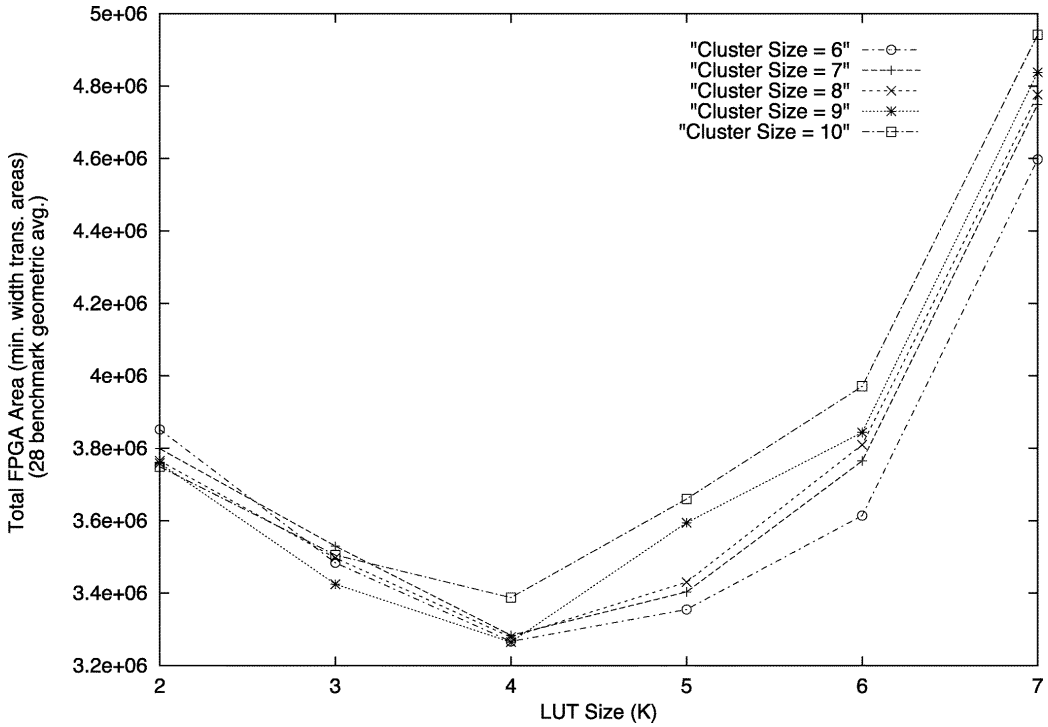
Figure 6.7b shows the number of BLEs on the critical path for every combination of K and N and compares to the only curve available from the reference study in Figure 6.7a. The curves decrease at the same rate. We also show that, in general, the number of BLEs on the critical path decreases as well with the size of the cluster (N).

Having similar area and delay curves comes as a validation of the conclusions of the previous study on parameterizable clusters. According to our results, the architectures that lead to the best area have (K, N) values of (4, 6) and (4, 9), while the best delay was observed for (K, N) values of (7, 5), (7, 8), (7, 7), (6, 5) and (6, 7), which concurs with the findings of Ahmed and Rose [2004].

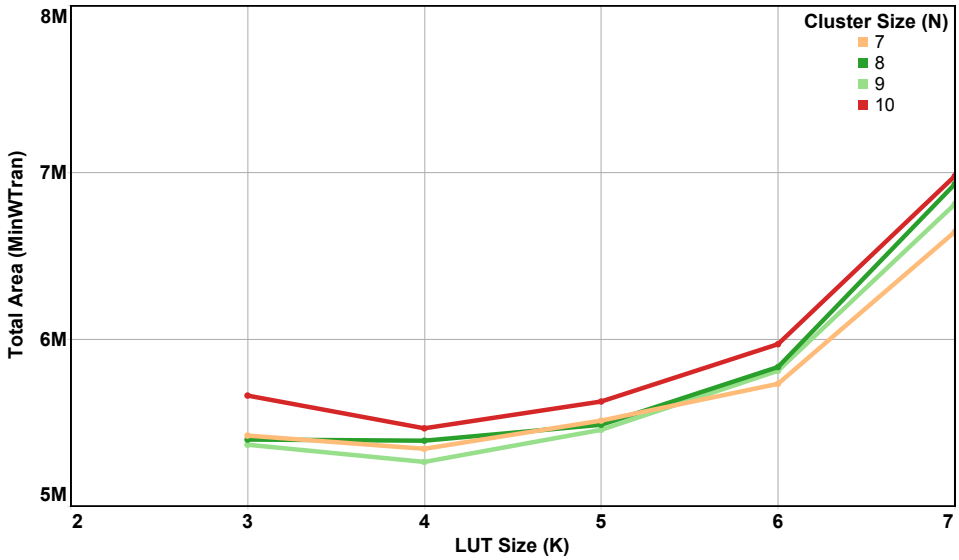
More importantly though, it is also a confirmation of the consistency and validity of FPRESSO's modeling. We were able to reach, with our automatic modeling approach, the very same conclusions of an architecture exploration with manually modeled circuits. With almost no effort and within a few hours, we were able to model all the architectures considered in this exploration.

6.4 Expanding the Exploration Space

With such a simple-to-use and fast architecture modeling technique that interfaces well with the FPGA CAD tools, wide-space explorations are now feasible for all FPGA architects. We demonstrate in this section how the boundaries of the previous explorations can be easily

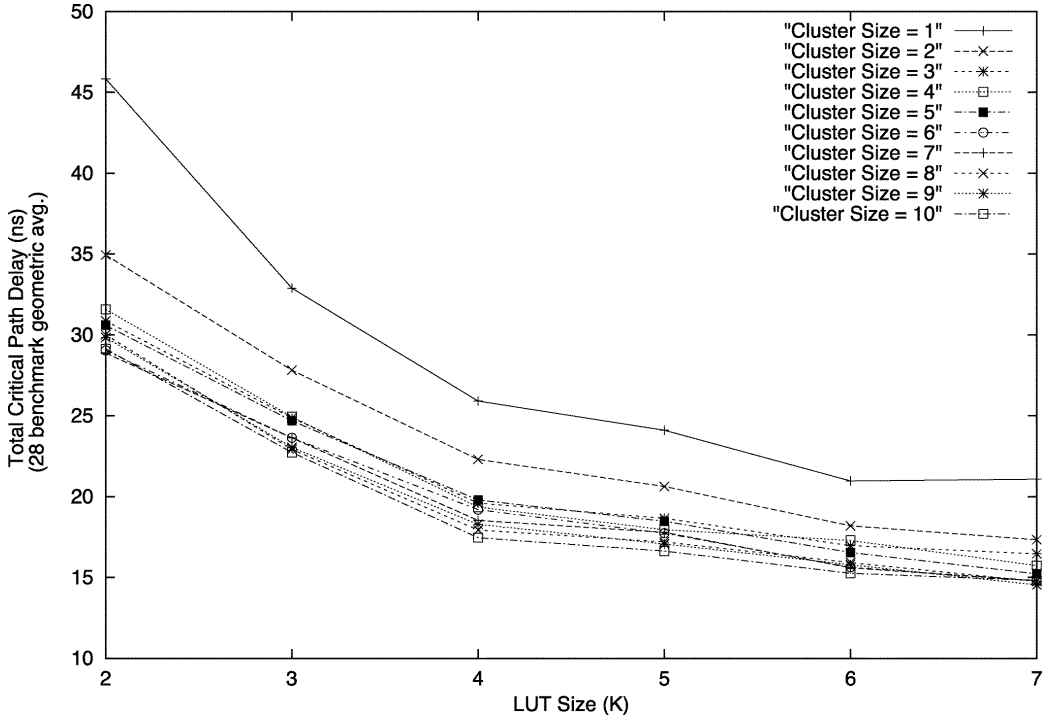


(a) Total area with respect to the LUT size (K), for large cluster sizes, from the reference study [Ahmed and Rose, 2004].

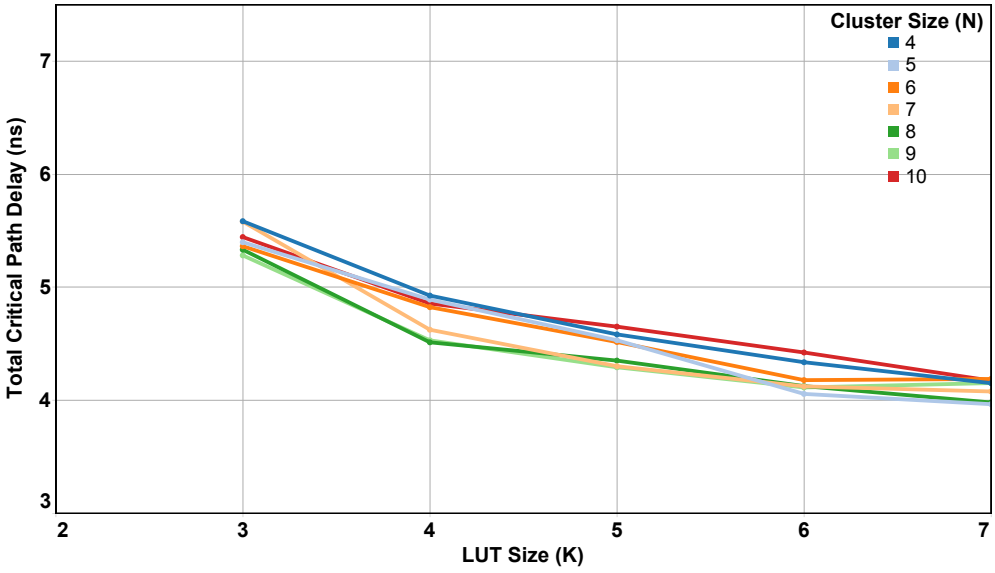


(b) Total area with respect to the LUT size (K), for large clusters of sizes (N) 7 to 10, measured using our methodology.

Figure 6.5 – Total area of the MCNC benchmarks with respect to the LUT size (K), for relatively large cluster sizes, as measured in both the reference study and our experiments.

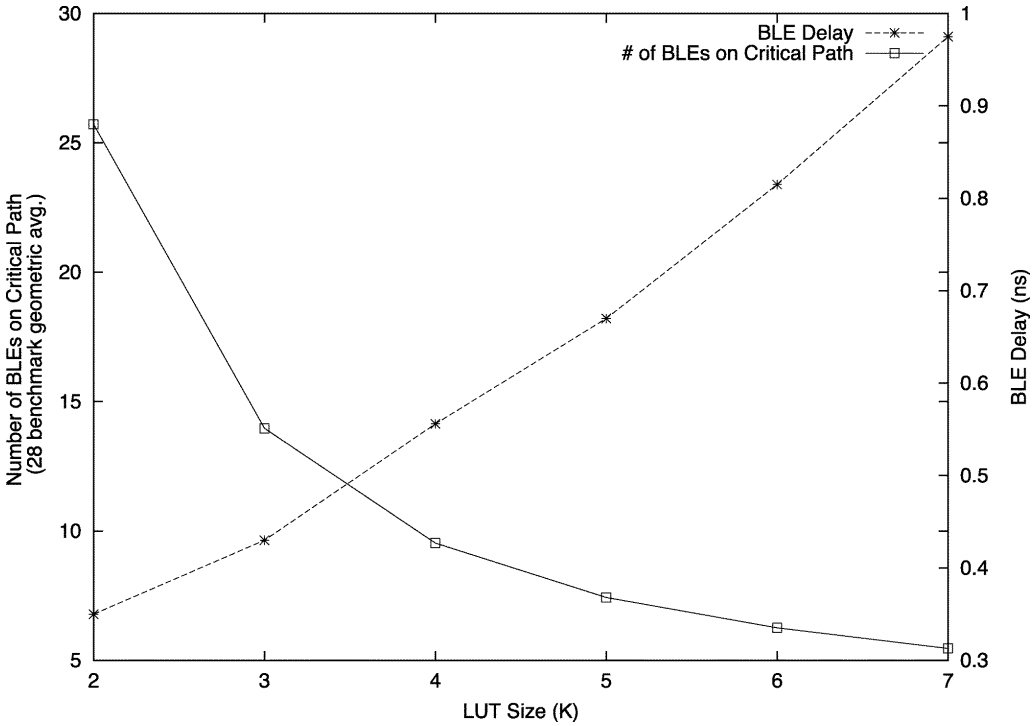


(a) The total delay (in ns) as reported by the reference study [Ahmed and Rose, 2004].

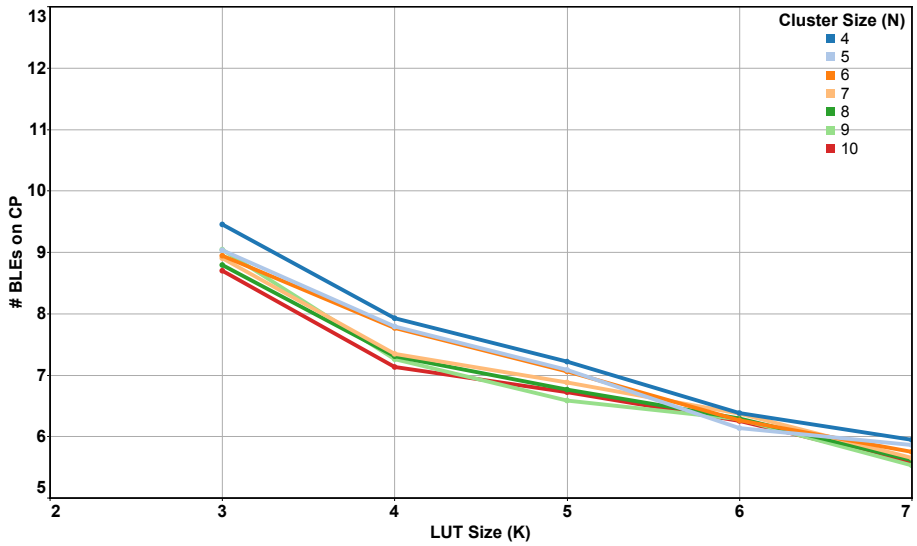


(b) The total delay (in ns) measured using our experimental methodology.

Figure 6.6 – Total delay (in ns) with respect to the LUT size K, (a) as reported by the reference study and (b) as measured from our experiments.



(a) The number of BLEs on the critical path as reported by the reference study [Ahmed and Rose, 2004].



(b) The number of BLEs on the critical path measured using our experimental methodology, for different cluster sizes (N).

Figure 6.7 – The number of BLEs on the critical path decreases as the LUT size (K) increases, both in the reference study and in our results. We also show that it generally decreases as well, as N increases.

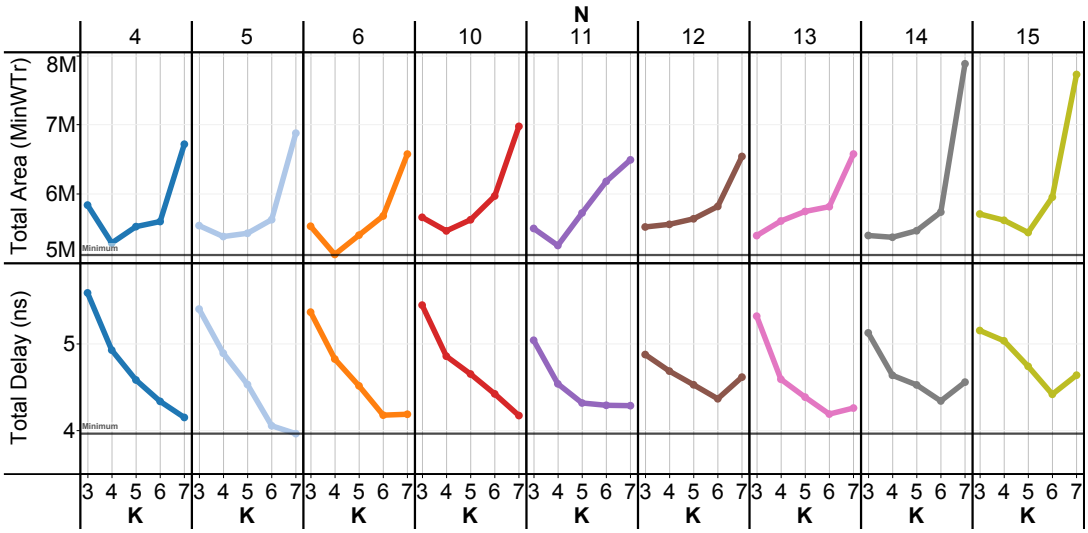


Figure 6.8 – The measured delay and area as the cluster size (N) varies, for different LUT sizes (K). In general, no clear trend emerges to favor large cluster sizes.

expanded to evaluate more and potentially new features in FPGA architectures: we explore the effect of larger cluster sizes, depopulated crossbars, and fracturable LUTs on the performance and area of the FPGAs.

6.4.1 Evaluating Large Clusters

Setting the maximum cluster size to 10 can be a limitation to the search space, especially since bigger clusters may seem promising: having more logic within the cluster can favor local feedbacks and reduce the use of global routing. Thus, we extend the experiments to a cluster size of 15.

Figure 6.8 shows the change in area and delay as N increases, for all K. In general, larger values of N are not particularly advantageous, neither in area nor in delay. However, there are some particular cases for which some area and/or delay improvement is observed. Thus, increasing the cluster size is not necessarily advantageous to all architectures but may present some improvement for particular cases.

6.4.2 Evaluating Crossbar Density

As the clusters get bigger, the crossbars can become very expensive in terms of both area and delay, as discussed in Section 2.1.3. Sparse crossbars were introduced as a compromise between the flexibility of the routing and the cost of the full density [Lemieux et al., 2000; Lemieux and Lewis, 2001]. By depopulating a crossbar, each of the crossbar's outputs can be connected to only a fraction ($F_{c_{local}}$) of its inputs. Therefore, using a sparse crossbar translates to smaller multiplexers that reduce the routing flexibility of the LUT inputs but, at the same

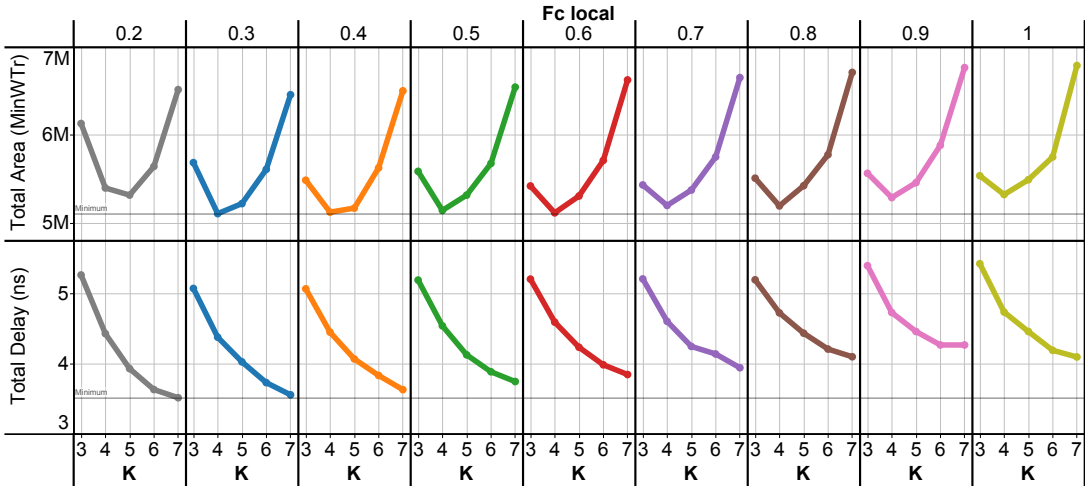


Figure 6.9 – The effect of sparse crossbars with different density $F_{c_{local}}$ on the delay and area, for multiple LUT inputs (K), averaged over all cluster sizes (N). This figure shows that the best results are achieved for the crossbar with 30% density.

time, reduce the area and critical path delay of the cluster.

With the new optimization algorithms in CAD tools (possibly able to better route under restricted resources) and at smaller technology nodes (affecting the delay cost of long wires and large fanout nodes versus the delay of LUTs), the effect of the crossbars and of their sparsity on the FPGA performance might change dramatically.

To study these effects, we evaluate the crossbar sparsity on a wide range of architectures by varying $F_{c_{local}}$ between 0.2 (i.e., each output can connect to 20% of the inputs) and 1 (i.e., a fully populated crossbar) for K between 3 and 7, and N between 4 and 10. FPRESSO is designed in a way that, given a certain number of inputs/output, if the crossbar density cannot be achieved, this density is increased until all inputs can be connected. So, although a 20% crossbar is considered in these experiments, in several cases, the effective density is slightly higher than 20% (by a few percentages). Furthermore, FPRESSO distributes, by default, the inputs of the sparse crossbars uniformly among the created multiplexers.

Figure 6.9 shows how the delay and area change with the density of the crossbar for the different K, averaged over N. The first observation is that, even after varying the density, our previous conclusions of Section 6.3 on the optimal LUT sizes for area (4 and 5) and delay (6 and 7) still clearly hold for most $F_{c_{local}}$. However, more importantly, as $F_{c_{local}}$ decreases, area also decreases, until it reaches its optimum at 30% density, before increasing back at 20%. Interestingly enough, the best delay is also observed at around 20% and 30% density, making this an architectural sweet spot. Hence, the best compromise between the size of the crossbar and the routing flexibility is generally achieved, for most architectures, at the low density of around 30%. It benefits from the small-sized multiplexers to improve the delay and area contribution of the crossbar while maintaining enough flexibility not to introduce excessive routing overhead.

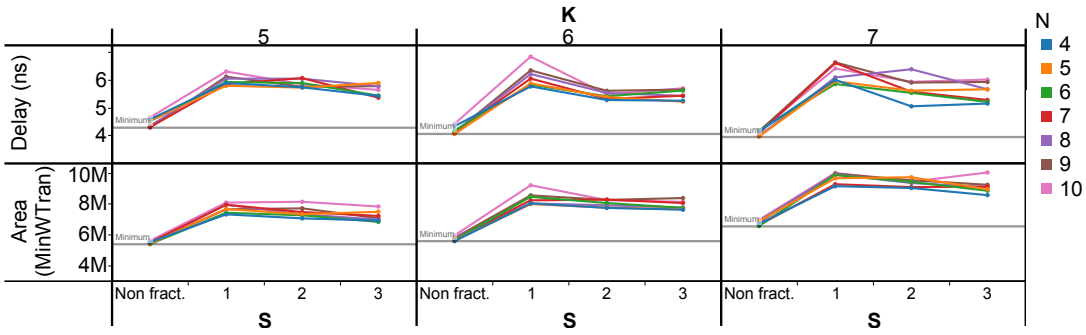


Figure 6.10 – Delay and area for architectures with fracturable K-LUTs and up to 3 shared inputs (S). The results for non-fracturable K-LUTs are also added for comparison.

6.4.3 Evaluating Fracturable LUTs

To evaluate the effect of the fracturable LUTs on the FPGA performance, we run experiments for LUTs of sizes 5, 6, and 7, where each LUT can be fractured into two smaller ones, of sizes 4, 5, and 6, respectively, with shared inputs. The number of shared inputs S is also added as a parameter and varied between 1 and 3. The architectures were generated for clusters of sizes $N \leq 10$ and with full crossbars to filter out the effect of sparse crossbars on the results. It is important to mention that we use, in these experiments, the default ABC version that comes with VTR 7.0, and map using the *if* command with the wiremap option enabled.

Figure 6.10 shows the delay and area for architectures with fracturable K-LUT, over multiple cluster sizes and for up to three shared inputs. The same results for non-fracturable K-LUT are also added for comparison. The results vary, depending on the selected parameters but, in general, one can observe a clear trend irrespective of the LUT or cluster size. When compared to the results of the non-fracturable architecture, fracturable LUTs do not bring any improvement. A slight increase in area and delay was to be expected: the BLE has more inputs in the fracturable architecture, which results in bigger and slightly slower crossbars, and the LUTs are also slower to support the fracturability. However, one would have thought that the additional flexibility of fracturable LUTs should have largely overcome this overhead—alas, this appears not to be the case.

To better understand the reasons behind this behavior, we evaluate the effect of the CAD tools on the results, by analyzing the outputs of the mapping, packing and routing stages as shows in Figure 6.11. Although this applies to any fracturable architecture, we take a cluster of size 10 with fracturable 5-LUTs ($K = 5$, $N = 10$). Any advantage the fractuable 5-LUT should bring is highly influenced by the number of 4-LUTs (and less) generated by the technology mapper and by their shared inputs that can create opportunities to efficiently utilize the 5-LUT. However, only about 18% of the total LUTs have four inputs which limits considerably these opportunities. Clearly, ABC is not aware that LUTs can be fractured and, as such, does not try to optimize its mapping accordingly. Furthermore, the packer is adding more damage by under-utilizing the available resources: almost half of the 5-LUTs are fractured into two

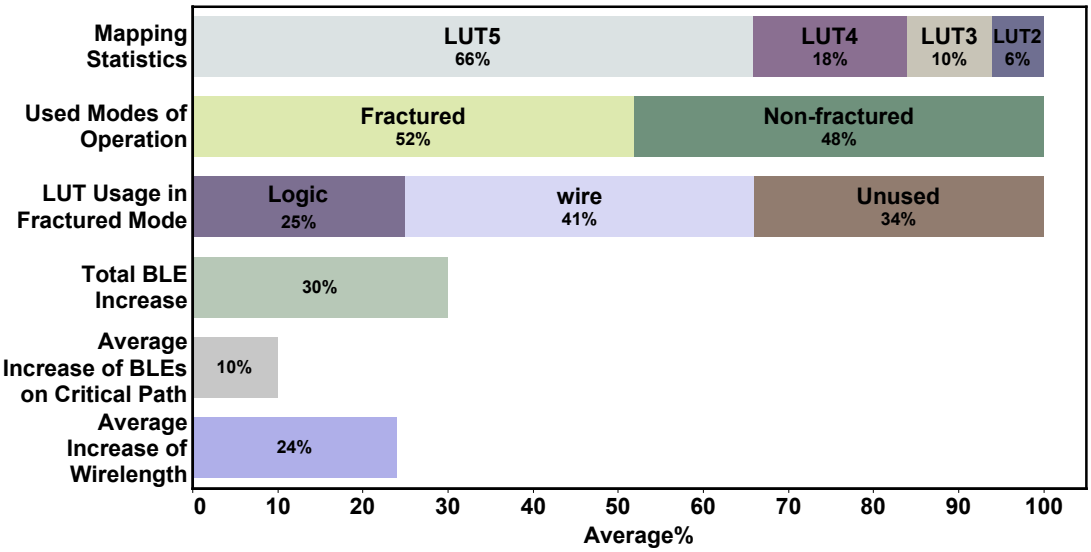


Figure 6.11 – Analysis of the mapping and packing results for a fracturable 5-LUT architecture with $N = 10$.

4-LUTs, and out of these 4-LUTs only 25% are used for logic, while 41% are used as wires and about 34% are completely unused. Using only 25% of the fractured LUTs is a major waste of resources and causes this overhead, mainly since it increases the total number of BLEs by 30% (compared to the non-fracturable architecture), the number of BLEs on the critical path by about 10% and the average wirelength by 24%. All this added together gives an idea of the reasons behind the deterioration of the results for fracturable LUTs and is an indication that the existing academic CAD tools are not properly equipped or designed, yet, to support this feature. It is not clear to us at this point how the commercial tools handle the fracturable LUTs and if they result in a better utilization of the available resources, but we plan on further investigating that in the future.

6.5 Discussion

After exploring over 1,200 FPGA architectures and running more than 41,000 benchmark simulations, we show that (i) the observations and conclusions of Ahmed and Rose's classic study of 2004 remain valid with our automatic architecture modeling approach, and that (ii) extremely large search spaces are now easily explorable with minimal effort and without any specific knowledge in transistor design.

Reaching the same conclusions as a well-established and industrially-plausible study validates the correctness of our approach and the relative accuracy of its modeled architectures.

In the process, we demonstrate that the MCNC benchmarks remain useful in such architectural explorations and have, in this context, similar behavior as the more complex VTR benchmarks.

We also show that a 30% density crossbar systematically achieves better results for most architectures, offering the best compromise between routing flexibility and cost. Additionally, we observe that, when using the academic CAD tools, the overhead of the fracturable LUTs overcomes any advantage they theoretically bring. We show, through an analysis at the different stages of the CAD flow, that the current tools do not fully leverage the potentials of the fracturable LUTs.

It would have been almost impossible to explore that many architectures, evaluate these several features, and reach these conclusions if the architecture modeling was not entirely automated, fast, and simple to use.

7 Conclusion

Retargetable toolchains are one of the keystones of architectural research. Supporting a wide range of easily-described and arbitrary architectures, these tools, like the VTR flow, are essential to any evaluation of the performance of new architectures on a set of applications.

However, to achieve sound architecture evaluations, FPGA CAD tools require reliable delay and area estimations of the routing and logic elements within the architecture. And, providing the tools with such estimations means generally designing, optimizing, and simulating the architecture at the transistor level. The caveat, however, is that typical FPGA architects do not necessarily have the required skill set nor the time to perform such a tedious and impractical task for hundreds of architectures. This major difficulty set some serious limitations on the FPGA architectural explorations and slowed down the research in that area.

7.1 Computer Architecture Analogy

Looking at other disciplines, one can notice, for instance, in computer architecture, the abundance of tools that support the researchers in their search and design of customized architectures. Architects have access to a variety of fairly accurate architectural simulators, customizable in many aspects [Burger and Sivasubramaniam, 2004]. Yet, such tools depend, much as VTR in the FPGA world, on users providing assumptions (typically on latency and cost) of each and every component they introduce or modify when they explore new architectural ideas.

A few years back, researchers noticed how difficult it was for most architects to predict the effect of some architectural changes, most notably in the memory hierarchy, on its area and latency: understanding most of the implications requires a deep knowledge of the transistor-level implementation options of leading-edge memories, which is clearly outside of the classic skill set of an architect. CACTI [Wilton and Jouppi, 1994] was born out of that need: an easy-to-use and sound model for caches and other memory hierarchy elements. Over a couple of decades, six major revisions [Muralimanohar et al., 2009], and continuous new extensions [Jouppi et al.,

2015], CACTI has helped literally hundreds of research groups in their scientific quest.

It has become increasingly obvious that FPGA architectural research suffers today from the same syndrome that afflicted in the early nineties the computer architectural community: namely, the difficulty of combining in the same researcher or even in the same research group acute architectural intuition and leading-edge transistor-level design skills.

7.2 Bringing the Concept to FPGAs

More modestly compared to the CACTI endeavor, we have shown a new path towards quick and efficient modeling of arbitrary FPGA architectures: we can generate within minutes optimized VTR models with reasonably faithful delay and area estimations.

Inspired from the semicustom flow, we prepare libraries of the basic building blocks for FPGA architectures, each with different drive strengths. The library will, for example, contain different transistor sizings of every K-LUT and variable-input multiplexers. Then, whenever an architecture is to be modeled, a classic optimizer constructs the circuit using cells from the library, with optimal drive strengths, and adds buffers wherever needed. By splitting the flow—performing all the transistor sizing and simulations offline, and only optimizing the architecture at runtime— we are able to achieve fast architecture modeling and enable express architectural explorations, with acceptable accuracy.

We show, by repeating a well known architecture exploration study, that the manual architecture optimization and our modeling approach can reach the same conclusions, although we do it with significantly less effort. Then, by extending the search space, we demonstrate how FPRESSO is capable of evaluating, quickly and easily, the effect of other features like depopulated crossbars and fracturable LUTs on the overall FPGA performance.

7.3 Meeting Industrial Standards

The work we present in this thesis addresses a pressing problem in FPGA architectural research by modeling arbitrary FPGA architectures, quickly and with reasonable accuracy, to enable sound and consistent architectural explorations.

Our work fills a substantial gap in the CAD flow for academic FPGA research and offers architects and all classes of FPGA users the means to easily and quickly evaluate new architectures. Naturally though, our work cannot compete with commercial FPGAs since, targeting mainly academic research, it does not yet meet the industrial standards. The architectures of these commercial FPGAs have evolved beyond simple reconfigurable logic to more complex structures with hard logic, DSP, and memory blocks. Thus, if we wish to model industrially competitive architectures, we need to extend our palette of supported logic.

Implementing arithmetic circuits on commercial FPGAs, for example, benefits from the ex-

isting hard adders and dedicated carry chains [Altera, 2014; Xilinx, 2014]. These chains allow for a fast propagation of the carry signals without having to go through the global routing of the FPGA fabric, offering major speedup of arithmetic operations. In general, our modeling approach is highly modular and can easy support hard logic elements, within the cluster, as long as they are included in the library. However, the challenge of adding carry chains to the modeled architecture resides in floorplanning the cluster in a way that prioritizes the minimization of these dedicated wires, to make sure that they are as fast as possible. In theory, this can be done by assigning weights to the wires so that the cost function of the floorplanning algorithm increases substantially with longer carry chain wires. Support for carry chains is probably one of the most pressing features that can be added to our modeling technique.

Furthermore, the most recent benchmark suites like the VTR [Rose et al., 2012] and Titan [Murray et al., 2013] benchmarks were introduced to emulate real applications that can target FPGAs. These circuits are generally bigger than the traditionally-used MCNC benchmarks [Yang, 1991] and include memory and DSP blocks. FPRESSO does not currently model DSP and memory blocks, but this is one of the features that we plan on adding, in the future, to our modeling technique. The most recent version of COFFE [Yazdanshenas et al., 2017] added the possibility of automatically generating and optimizing Block RAMs (BRAMs) for both SRAM and Magnetic Tunneling Junction technologies. Although this needs careful investigation, we believe that we can leverage these new features in COFFE to optimize memory blocks for our library of cells.

Perhaps one of the strongest motivations we had in finding a solution to the architecture modeling problem was to be able to explore wildly unconventional architectures and to specifically identify the optimal FPGA architecture with the And-Inverter Cones as basic logic elements [Zgheib et al., 2015, 2014]. So it is only logical that our next step would be to include the AICs into our library of macrocells. This requires adding AIC support into the customized version of COFFE to automatically generate a variety of cells for the characterization and integration in the library. However, being a multi-output logic element, we suspect that the AICs will introduce a particular and new set of challenges to our library generation flow; but this will be carefully studied in the future.

7.4 A Stepping Stone

FPGAs have gone from a minimalistic market in the eighties to a multi-billion business in which the top semiconductor vendors seem to place hopes for the future of computing. Yet, their fundamental architecture has not changed much in decades. It is high time for academics to look afresh at the FPGA architecture and this thesis lays a small stone to help researchers perform new and adventurous explorations.

Bibliography

ABC (n.d.). *ABC: A System for Sequential Synthesis and Verification*. Berkeley Logic Synthesis and Verification Group, Berkeley, Calif. http://www.eecs.berkeley.edu/~alanmi/abc/.

Adya, S. N. and Markov, I. L. (2003). Fixed-outline floorplanning: enabling hierarchical design. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 11(6):1120–1135.

Ahmed, E. and Rose, J. (2000). The effect of LUT and cluster size on deep-submicron FPGA performance and density. In *Proceedings of the 2000 ACM/SIGDA 8th International Symposium on Field Programmable Gate Arrays*, pages 3–12, Monterey, Calif.

Ahmed, E. and Rose, J. (2004). The effect of LUT and cluster size on deep-submicron FPGA performance and density. *IEEE Transactions on Very Large Scale Integration Systems*, 12(3):288–298.

Altera (2014). *Stratix V Device Handbook, vols. 1 and 2*. Altera Corporation. `http://www.altera.com/literature/`.

Betz, V., Rose, J., and Marquardt, A. (1999). *Architecture and CAD for deep-submicron FPGAs*. Kluwer Academic, Boston, Mass.

Burger, D. and Sivasubramaniam, A. (2004). Tools for computer architecture research. *SIGMETRICS Performance Evaluation Review*, 31(4):2–3.

Chandy, J. A. and Banerjee, P. (1996). Parallel simulated annealing strategies for VLSI cell placement. In *Proceedings of the 9th International Conference on VLSI Design: VLSI in Mobile Communication*, pages 37–42.

Chang, Y., Chang, Y., Wu, G., and Wu, S. (2000). B*-Trees: a new representation for non-slicing floorplans. In *Proceedings of the 37th ACM/IEEE Design Automation Conference*, pages 458–463.

Chen, T. and Chang, Y. (2006). Modern Floorplanning Based on B*-Tree and Fast Simulated Annealing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(4):637–650.

Chiasson, C. and Betz, V. (2013). COFFE: Fully-automated transistor sizing for FPGAs. In *Proceedings of the IEEE International Conference on Field Programmable Technology*, pages 34–41.

Chinnery, D. and Keutzer, K. (2002). *Closing the Gap Between ASIC & Custom: Tools and Techniques for High-Performance ASIC Design*. Springer US, New York, NY.

Chu, C. and Wong, Y. C. (2008). FLUTE: Fast Lookup Table Based Rectilinear Steiner Minimal Tree Algorithm for VLSI Design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(1):70–83.

Conn, A. R., Coulman, P. K., Haring, R. A., Morrill, G. L., Visweswariah, C., and Wu, C. W. (1998). JiffyTune: circuit optimization using time-domain sensitivities. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(12):1292–1309.

Conn, A. R., Elfadel, I. M., Molzen, W. W., O'Brien, P. R., Strenski, P. N., Visweswariah, C., and Whan, C. B. (1999). Gradient-based optimization of custom circuits using a static-timing formulation. In *Proceedings of the 1999 Design Automation Conference*, pages 452–459.

Elmore, W. C. (1948). The Transient Response of Damped Linear Networks with Particular Regard to Wideband Amplifiers. *Journal of Applied Physics*, 19(1):55–63.

Eriksson, H., Larsson-Edefors, P., Henriksson, T., and Svensson, C. (2003). Full-custom vs. standard-cell design flow: an adder case study. In *Proceedings of the Asia and South Pacific Design Automation Conference*, pages 507–510.

Fang, J., Chang, Y., Chen, C., Liang, W., Hsieh, T., Satria, M. T., and Han, C. (2009). A parallel simulated annealing approach for floorplanning in VLSI. In *Proceedings of the 9th International Conference on Algorithms and Architectures for Parallel Processing*, pages 291–302.

Fishburn, J. P. and Dunlop, A. E. (1985). TILOS: A posynomial programming approach to transistor sizing. In *Proceedings of the International Conference on Computer Aided Design*, pages 326–328.

Greene, J., Kaptanoglu, S., Feng, W., Hecht, V., Landry, J., Li, F., Krouglyanskiy, A., Morosan, M., and Pevzner, V. (2011). A 65nm flash-based FPGA fabric optimized for low cost and power. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 87–96.

Guo, P., Takahashi, T., Cheng, C., and Yoshimura, T. (2001). Floorplanning using a tree representation. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 20(2):281–289.

Guo, P.-N., Cheng, C.-K., and Yoshimura, T. (1999). An O-tree Representation of Non-slicing Floorplan and its Applications. In *Proceedings of the 36th Annual ACM/IEEE Design Automation Conference*, pages 268–273.

Hwang, F., Richards, D., and Winter, P. (1992). *The Steiner Tree Problem*. Annals of Discrete Mathematics. Elsevier Science.

Jiang, Z., Zgheib, G., Yu Lin, C., Novo, D., Yang, L., Huang, Z., Yang, H., and Ienne, P. (2015). A technology mapper for depth-constrained FPGA logic cells. In *Proceedings of the 25th International Conference on Field-Programmable Logic and Applications*, pages 1–8, London.

Jouppi, N. P., Kahng, A. B., Muralimanohar, N., and Srinivas, V. (2015). CACTI-IO: CACTI with OFF-chip power-area-timing models. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, VLSI-23(7):1254–67.

Kaptanoglu, S., Bakker, G., Kundu, A., Corneillet, I., and Ting, B. (1999). A new high density and very low cost reprogrammable FPGA architecture. In *Proceedings of the 1999 ACM/SIGDA Seventh International Symposium on Field Programmable Gate Arrays*, pages 3–12.

Kasamsetty, K., Ketkar, M., and Sapatnekar, S. S. (2000). A new class of convex functions for delay modeling and its application to the transistor sizing problem [CMOS gates]. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(7):779–788.

Kim, J. H. and Anderson, J. H. (2015). Synthesizable FPGA fabrics targetable by the Verilog-to-Routing (VTR) CAD flow. In *Proceedings of the 25th International Conference on Field-Programmable Logic and Applications*, pages 1–8.

Kirkpatrick, S., Gelatt, C. D., and Vecchi, M. P. (1983). Optimization by simulated annealing. *Science*, 220(4598):671–680.

Kuon, I. and Rose, J. (2011). Exploring area and delay tradeoffs in FPGAs with architecture and automated transistor design. *IEEE Transactions on Very Large Scale Integration Systems*, 19(1):71–84.

Kuon, I., Tessier, R., and Rose, J. (2008). *FPGA Architecture: Survey and Challenges*. Now, Delft, The Netherlands.

Lemieux, G., Leventis, P., and Lewis, D. (2000). Generating highly-routable sparse crossbars for PLDs. In *Proceedings of the 8th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 155–64, Monterey, Calif.

Lemieux, G. and Lewis, D. (2001). Using sparse crossbars within LUT clusters. In *Proceedings of the 2001 ACM/SIGDA Ninth International Symposium on Field Programmable Gate Arrays*, pages 59–68, Monterey, California, USA.

Lewis, D., Ahmed, E., Cashman, D., Vanderhoek, T., Lane, C., Lee, A., and Pan, P. (2009). Architectural enhancements in Stratix-III™and Stratix-IV™. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 33–42, New York, NY.

Lewis, D., Betz, V., Jefferson, D., Lee, A., Lane, C., Leventis, P., Marquardt, S., McClintock, C., Pedersen, B., Powell, G., Reddy, S., Wysocki, C., Cliff, R., and Rose, J. (2003). The stratix routing and logic architecture. In *Proceedings of the 2003 ACM/SIGDA Eleventh International Symposium on Field Programmable Gate Arrays*, pages 12–20, Monterey, Calif.

Lewis, D. et al. (2005). The Stratix II logic and routing architecture. In *Proceedings of the 13th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 14–20, Monterey, Calif.

Luu, J., Goeders, J., Wainberg, M., Somerville, A., Yu, T., Nasartschuk, K., Nasr, M., Wang, S., Liu, T., Ahmed, N., Kent, K. B., Anderson, J., Rose, J., and Betz, V. (2014a). VTR 7.0: Next generation architecture and CAD system for FPGAs. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 7(2):6:1–6:30.

Luu, J., McCullough, C., Wang, S., Huda, S., Yan, B., Chiasson, C., Kent, K. B., Anderson, J., Rose, J., and Betz, V. (2014b). On hard adders and carry chains in FPGAs. In *Proceedings of the 22nd IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 52–9, Boston, Mass.

Muralimanohar, N., Balasubramonian, R., and Jouppi, N. P. (2009). CACTI 6.0: A tool to model large caches. Technical Report HPL-2009-85, Hewlett-Packard Development Company, Palo Alto, Calif.

Murata, H., Fujiyoshi, K., Nakatake, S., and Kajitani, Y. (1995). Rectangle-packing-based Module Placement. In *Proceedings of the 1995 IEEE/ACM International Conference on Computer-aided Design*, pages 472–479.

Murray, K. E., Whitty, S., Liu, S., Luu, J., and Betz, V. (2013). Titan: Enabling large and complex benchmarks in academic CAD. In *Proceedings of the 23rd International Conference on Field-Programmable Logic and Applications*, pages 1–8.

Ousterhout, J. K. (1984). Switch-level delay models for digital MOS VLSI. In *Proceedings of the 21st Design Automation Conference*, pages 542–548.

Padalia, K., Fung, R., Bourgeault, M., Egier, A., and Rose, J. (2003). Automatic transistor and physical design of FPGA tiles from an architectural specification. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 164–172.

Parandeh-Afshar, H., Benbihi, H., Novo, D., and Ienne, P. (2012). Rethinking FPGAs: Elude the flexibility excess of LUTs with And-Inverter Cones. In *Proceedings of the 20th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 119–28, Monterey, Calif.

Parandeh-Afshar, H., Zgheib, G., Novo, D., Purnaprajna, M., and Ienne, P. (2013). Shadow And-Inverter Cones. In *Proceedings of the 23rd International Conference on Field-Programmable Logic and Applications*, pages 1–4, Porto, Portugal.

Rose, J., Francis, R. J., Lewis, D., and Chow, P. (1990). Architecture of field-programmable gate arrays: the effect of logic block functionality on area efficiency. *IEEE Journal of Solid-State Circuits*, 25(5):1217–1225.

Rose, J., Luu, J., Yu, C. W., Densmore, O., Goeders, J., Somerville, A., Kent, K. B., Jamieson, P., and Anderson, J. (2012). The VTR project: architecture and CAD for FPGAs from Verilog to routing. In *Proceedings of the 20th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 77–86.

Rubinstein, J., Penfield, P., and Horowitz, M. A. (1983). Signal Delay in RC Tree Networks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2(3):202–211.

Sechen, C. and Sangiovanni-Vincentelli, A. (1986a). The timberwolf placement and routing package. *IEEE Journal for Solid State Circuits*, SC-20:510–522.

Sechen, C. and Sangiovanni-Vincentelli, A. (1986b). Timberwolf3.2: A new standard cell placement and global routing package. In *Proceedings of the 23rd ACM/IEEE Design Automation Conference*, pages 432–439.

Sherwani, N. A. (1995). *Algorithms for VLSI Physical Design Automation*. Kluwer Academic Publishers, Norwell, MA, 2nd edition.

Singh, S., Rose, J., Chow, P., and Lewis, D. (1992). The effect of logic block architecture on FPGA performance. *IEEE Journal of Solid-State Circuits*, 27(3):281–287.

Smith, A. M., Wilton, S. J., and Das, J. (2009). Wirelength modeling for homogeneous and heterogeneous FPGA architectural development. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '09, pages 181–190.

Stoobandt, D. (2001). Multi-terminal nets do change conventional wire length distribution models. In *Proceedings of the 2001 International Workshop on System-level Interconnect Prediction*, SLIP '01, pages 41–48.

Wilton, S. J. and Jouppi, N. P. (1994). An enhanced access and cycle time model for on-chip caches. Technical Report WRL-93-5, Digital Equipment Corporation, Palo Alto, Calif.

Wong, D. F. and Liu, C. L. (1986). A New Algorithm for Floorplan Design. In *Proceedings of the 23rd ACM/IEEE Design Automation Conference*, pages 101–107.

Wu, G., Chang, Y., and Chang, Y. (2003). Rectilinear block placement using B*-Trees. *ACM Transactions on Design Automation of Electronic Systems*, 8(2):188–202.

Xilinx (2014). *7 Series FPGAs Configurable Logic Block*. Xilinx Inc. Version 1.7, `http://www.xilinx.com/`.

Xilinx (n.d.). *Virtex-5 User Guide*. Xilinx Inc. `http://www.xilinx.com/`.

Yang, S. (1991). Logic synthesis and optimization benchmarks user guide, version 3.0. Technical report, Microelectronics Center of North Carolina, Research Triangle Park, N.C.

Yazdanshenas, S., Tatsumura, K., and Betz, V. (2017). Don't Forget the Memory: Automatic Block RAM Modelling, Optimization, and Architecture Exploration. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 115–124.

Zgheib, G. and Ienne, P. (2016). Automatic wire modeling to explore novel FPGA architectures. In *Proceedings of the IEEE International Conference on Field Programmable Technology*, pages 181–184.

Zgheib, G. and Ienne, P. (2017). Evaluating FPGA clusters under wide ranges of design parameters. In *Proceedings of the 27th International Conference on Field-Programmable Logic and Applications*, Belgium.

Zgheib, G., Lortkipanidze, M., Owaida, M., Novo, D., and Ienne, P. (2016). FPRESSO: Enabling express transistor-level exploration of FPGA architectures. In *Proceedings of the 24th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 80–89, Monterey, Calif.

Zgheib, G., Parandeh-Afshar, H., Novo, D., and Ienne, P. (2015). And-inverter cones. In Gaillardon, P.-E., editor, *Reconfigurable Logic: Architecture, Tools, and Applications*, chapter 5, pages 127–48. CRC.

Zgheib, G., Yang, L., Huang, Z., Novo Bruna, D., Parandeh-Afshar, H., Yang, H., and Ienne, P. (2014). Revisiting And-Inverter Cones. In *Proceedings of the 22nd ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 45–54, Monterey, Calif.

Curriculum Vitae

Research Interests

FPGA architecture, CAD tools, architecture modeling and design.

Education

2011–2017 **Ph.D. in Computer Science**
School of Computer and Communication Sciences
Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland
Thesis: Leading the Blind: Automated Transistor-Level Modeling for FPGA Architects
Supervisor: Prof. Paolo Ienne

2009–2011 **Masters of Science in Computer Engineering**
Department of Electrical and Computer Engineering, School of Engineering
Lebanese American University, Byblos, Lebanon

2004–2009 **Bachelor of Engineering in Computer Engineering**
Department of Electrical and Computer Engineering, School of Engineering
Lebanese American University, Byblos, Lebanon

Publications

Conference Papers

Grace Zgheib and Paolo Ienne. "Evaluating FPGA clusters under wide ranges of design parameters". In: *Proceedings of the 27th International Conference on Field-Programmable Logic and Applications*. Ghent, Belgium, Sept. 2017.

Zhufei Chu, Xifan Tang, Mathias Soeken, Ana Petkovska, **Grace Zgheib**, Luca Amarù, Yinshui Xia, Paolo Ienne, Giovanni De Micheli, and Pierre-Emmanuel Gaillardon. "Improving circuit mapping performance through MIG-based synthesis for carry chains". In: *Proceed-*

ings of the 27th ACM Great Lakes Symposium on VLSI. Banff, Alberta, Canada, May 2017, pp. 131–36.

Zhihong Huang, Xing Wei, **Grace Zgheib**, Wei Li, Yu Lin, Zhenghong Jiang, Kaihui Tu, Paolo Ienne, and Haigang Yang. "Nand-nor: A compact, fast, and delay balanced FPGA logic element". In: *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '17. Monterey, Calif., 2017, pp. 135–140.

Grace Zgheib and Paolo Ienne. "Automatic wire modeling to explore novel FPGA architectures." In: *Proceedings of the IEEE International Conference on Field Programmable Technology*. Dec. 2016, pp. 181–184.

Grace Zgheib, Manana Lortkipanidze, Muhsen Owaida, David Novo, and Paolo Ienne. "Fpresso: Enabling express transistor-level exploration of FPGA architectures". In: *Proceedings of the 24th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. **Best Paper Award**. Monterey, Calif., Feb. 2016, pp. 80–89.

Ana Petkovska, **Grace Zgheib**, David Novo, Muhsen Owaida, Alan Mishchenko, and Paolo Ienne. "Improved carry-chain mapping for the VTR flow". In: *Proceedings of the 2015 International Conference on Field Programmable Technology*. Queenstown, New Zealand, Dec. 2015, pp. 80–87.

Zhenghong Jiang, **Grace Zgheib**, Colin Yu Lin, David Novo, Liqun Yang, Zhihong Huang, Haigang Yang, and Paolo Ienne. "A technology mapper for depth-constrained FPGA logic cells". In: *Proceedings of the 25th International Conference on Field-Programmable Logic and Applications*. London, Sept. 2015, pp. 1–8.

H. Harmanani, Danielle Azar, **Grace Zgheib**, and David Kozhaya. "An ant colony optimization heuristic to optimize prediction of stability of object-oriented components". In: *Proceedings of the 2015 IEEE International Conference on Information Reuse and Integration*. Aug. 2015, pp. 225–228.

Grace Zgheib, Liqun Yang, Zhihong Huang, David Novo Bruna, Hadi Parandeh-Afshar, Haigang Yang, and Paolo Ienne. "Revisiting And-Inverter Cones". In: *Proceedings of the 22nd ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. Monterey, Calif., Feb. 2014, pp. 45–54.

Hadi Parandeh-Afshar, **Grace Zgheib**, David Novo, Madhura Purnaprajna, and Paolo Ienne. "Shadow And-Inverter Cones". In: *Proceedings of the 23rd International Conference on Field-Programmable Logic and Applications*. Porto, Portugal, Sept. 2013, pp. 1–4.

Hadi Parandeh-Afshar, **Grace Zgheib**, Philip Brisk, and Paolo Ienne. "Reducing the pressure on routing resources of FPGAs with generic logic chains". In: *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. Monterey, Calif., Feb. 2011, pp. 237–46.

JOURNALS

Grace Zgheib and Iyad Ouaiss. “Enhanced technology mapping for FPGAs with exploration of cell configurations”. In: *Journal of Circuits, Systems and Computers* 24.3 (Mar. 2015).

BOOK CHAPTERS

Grace Zgheib, Hadi Parandeh-Afshar, David Novo, and Paolo Ienne. “And-Inverter Cones”. In: *Reconfigurable Logic: Architecture, Tools, and Applications*. Ed. by Pierre-Emmanuel Gaillardon. CRC, 2015. Chap. 5, pp. 127–48.

PATENTS

Hadi Parandeh-Afshar, David Novo Bruna, Paolo Ienne Lopez, and **Grace Zgheib**. *Non-LUT Field-Programmable Gate Arrays*. US Patent US9,231,594. Jan. 2016.