Towards Interconnect-Adaptive Packing for FPGAs

Jason Luu, Jonathan Rose, and Jason Anderson

Dept. Electrical and Computer Engineering, University of Toronto
Toronto, Ontario, Canada
vpr@eecg.utoronto.ca

ABSTRACT

In order to investigate new FPGA logic blocks, FPGA architects have traditionally needed to customize CAD tools to make use of the new features and characteristics of those blocks. The software development effort necessary to create such CAD tools can be a time-consuming process that can significantly limit the number and variety of architectures explored. Thus, architects want flexible CAD tools that can, with few or no software modifications, explore a diverse space. Existing flexible CAD tools suffer from impractically long runtimes and/or fail to efficiently make use of the important new features of the logic blocks being investigated. This work is a step towards addressing these concerns by enhancing the packing stage of the open-source VTR CAD flow [17] to efficiently deal with common interconnect structures that are used to create many kinds of useful novel blocks. These structures include crossbars, carry chains, dedicated signals, and others. To accomplish this, we employ three techniques in this work: *speculative packing*, *pre-packing*, and *interconnect-aware pin counting*. We show that these techniques, along with three minor modifications, result in improvements to runtime and quality of results across a spectrum of architectures, while simultaneously expanding the scope of architectures that can be explored. Compared with VTR 1.0 [17], we show an average 12-fold speedup in packing for fracturable LUT architectures with 20% lower minimum channel width and 6% lower critical path delay. We obtain a 6 to 7-fold speedup for architectures with non-fracturable LUTs and architectures with depopulated crossbars. In addition, we demonstrate packing support for logic blocks with carry chains.

Categories and Subject Descriptors

B.5.2 [**Design Aids**]: Automatic Synthesis, Optimization

Keywords

FPGA; Algorithms; Packing; Clustering; Architecture

1. INTRODUCTION

The architecture of an FPGA logic block has a significant impact on the overall performance, area, and power consumption of circuits implemented on the device. Over the years, market demands and technological innovations have driven the evolution of FPGA logic blocks from simple groups of LUTs and flip-flops [2] to more complex blocks such as memories with configurable aspect ratios [21] [1], multipliers with selectable size and quantity [1], and general-purpose logic blocks with different modes of operation. As the economics of technology scaling continues to drive more applications towards programmable devices, we expect new ideas to arise in the architecture of FPGA logic blocks to meet the demands of a changing market.

As FPGA architects seek to explore increasingly sophisticated logic blocks, the difficulty of conducting experiments to evaluate their quality has correspondingly increased. One reason for this difficulty is that many CAD tools employed in these experiments have restrictive architectural assumptions that require significant changes before any evaluation work can be done. The release of VTR 1.0 [17], an open source FPGA architecture exploration CAD flow, was a recent effort by Rose et al. towards addressing this problem by greatly expanding the scope of FPGA architectures that can be targetted without software changes. However, this flexibility created new challenges in the packing stage of the FPGA CAD flow. Packing assigns a technology-mapped user netlist to the various physical logic blocks of a target FPGA architecture. To support a greater variety of logic blocks, pack time in VTR 1.0 increased to the point where it rivals placement time on the VTR heterogeneous architectures. For more complex architectures, pack time in VTR can regularly exceed place-and-route time [14]. Our work seeks to address this runtime issue while simultaneously expanding the scope of architectures that can be explored. We first describe precisely why the VTR packer is slow before describing our approach to this problem.

VTR provides the architect great freedom when specifying logic block architectures including the ability to specify any arbitrary interconnect structure within a logic block. Support for arbitrary interconnect enables the natural expression of a wide range of architectural constructs. These include carry chains, crossbars, optionally registered inputs/outputs, and control signals, which can be expressed by simply stating how various components are connected together. However, this level of customization creates a computationally challenging packing problem. The packing algorithm must determine if the internal connectivity within a logic block can successfully route the sections of the netlist that are assigned into that logic block. The packing algorithm in VTR [9], as with other prior attempts on supporting arbitrary interconnect [18] [20], employs heavy use of detailed routing to check for routability. This results in a packer that is often unnecessarily slow.

Our goal is to develop a packing tool and algorithm that runs quickly for architectures with simple interconnect, spends medium computational effort on architectures with moderately complex interconnect, and only uses heavy computational effort on architectures with very complex interconnect. Our approach is to automatically use a faster, simpler algorithm when interconnect structures that are easier to deal with are encountered. For example, if an architecture contains full crossbars, then computationally intensive routing checks within the logic block are not necessary because routing is guaranteed as long as the number of pins to be connected is below a certain threshold. Similarly, if an architecture has an inflexible carry chain, then we know that the blocks that form that chain must be kept together in a strict order.

In this work, we enhance the packing stage of VTR [17]. We enable the packer in VTR to adapt computational effort based on architect-specified interconnect through three techniques: First, *speculative packing* attempts to save runtime by optimistically skipping detailed legality checks at intermediate steps and then checking all legality rules after a logic block is full. Second, *pre-packing* groups together netlist blocks that should stay together as one unit during packing. This helps the packer deal with interconnect structures with limited or no flexibility, such as carry chains and registered input/output pins. Third, *interconnect-aware pin counting* reduces the more complex routing problem to a simple counting problem, which is inferred from the architecture.

2. PRIOR WORK

Much of the prior work in packing focused on simple logic blocks that consist only of LUTs and flip-flops interconnected by full crossbar interconnect [12] [19] [3] [8] [7] [4].

There have been several attempts to make packers that have more general types of interconnect within the logic block. Ni [15] proposed a tool that targets the same simple style logic block described above but with arbitrary logic elements instead of just pairs of LUTs and flip-flops. Wang [20] and Lemieux [6] proposed different ways to target logic blocks with depopulated crossbars. Cong proposed a tool, RASP, where a variety of different interconnect in a logic block is permitted but the user is then responsible for creating custom heuristics to map to that logic block [5]. Paladino [16] proposed a packer that focuses on modelling control signals and carry chains within a logic block. All these methods provide point solutions to specific constructs, but do not give a general, comprehensive strategy towards automatically handling arbitrary interconnect structures.

A few related works have attempted to model arbitrary interconnect in logic blocks. Sharma [18] proposed a placement algorithm that can explore FPGAs with arbitrary general interconnect. That work could be extended to target arbitrary interconnect within a logic block. Sharma's approach is to regularly sample the underlying interconnect during annealing-based placement by employing detailed routing repeatedly. However, this approach results in extremely long runtime. Similar to Sharma's work but in the context of FPGA packing, Luu [9] proposed a greedy packer, called AAPack, that employed detailed routing to check the legality of all intermediate (partial) packing solutions. Although Luu benefited from a much smaller routing problem, namely working at the logic block level, Luu's tool still ran two orders of magnitude slower than T-VPack [12] on the simple FPGA architectures. AAPack 6.0 is the current packer used in VTR 1.0.

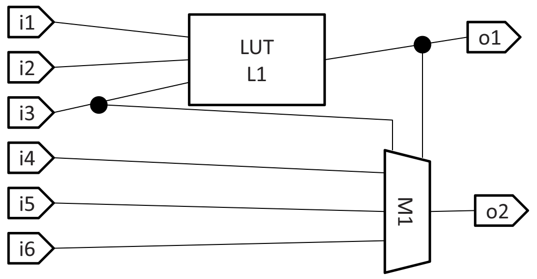

Figure 1: Example of a netlist with a LUT, a mux, and some I/O pads.

3. PROBLEM DEFINITION

The inputs to packing are a technology-mapped logical netlist and a description of the physical logic blocks of a target architecture [9]. The output of packing is a netlist of physical logic blocks that implement the input user netlist. This netlist of logic blocks is then placed and routed in the subsequent stages of the FPGA CAD flow.

A technology-mapped netlist is a flattened view of a user circuit. It consists of blocks, called *atoms*, that are connected together by *nets*. Examples of atoms include LUTs, flip-flops, memory slices, and I/Os. Atoms are classified based on what kind of functionality they represent. We call this the atom's *logic model*. During packing, each atom will be mapped to a unique physical unit, called a *primitive*, within one of the logic blocks. Fig. 1 shows an example of a netlist. This netlist consists of one atom of logic model *LUT*, one atom of logic model *mux*, six atoms of logic model *input pad*, and two atoms of logic model *output pad*.

The target architecture defines the different types of logic blocks available. A logic block definition includes the primitives that exist in the block, a hierarchical description of how those primitives are organized, and the routing structures within the logic block. Each type of logic block can appear in the FPGA in different quantities. For example, there may be one configurable multiplier block for every four soft logic blocks.

The primitives within a logic block are organized in a hierarchy. At the top of the hierarchy is the logic block itself. Below the logic block, the architect can specify any arbitrary tree hierarchy of *subclusters* and primitives. A subcluster is a node in the logic block hierarchy that can contain other subclusters or primitives.

The interconnect within a logic block provides connectivity between subclusters, primitives, and logic block input/output pins. In this expanded definition of the packing problem, the architect is allowed to specify any arbitrary interconnect network within a logic block. This extension enables a far more powerful expression of different architec-

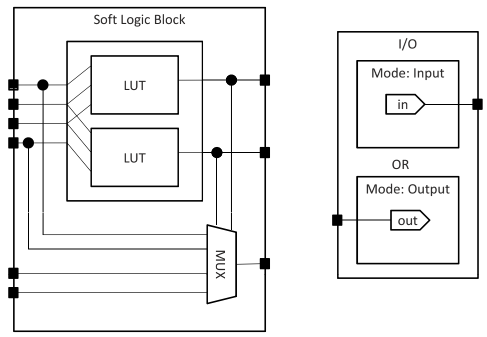

Figure 2: Example of an architecture with I/Os and soft logic blocks.

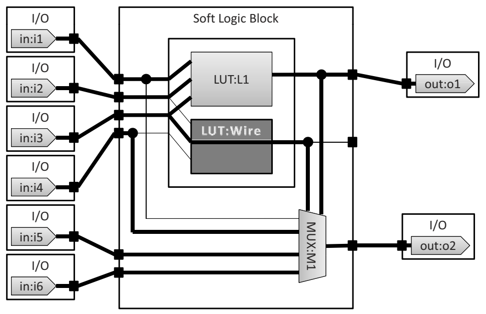

Figure 3: Example of a packing solution mapping a netlist to a set of logic blocks

tures but does create a potentially more difficult packing problem.

Structures in modern logic blocks can have different *modes*, which are mutually exclusive states of the logic; for example, a fracturable LUT can either be in its large, unfractured state, or be two smaller, "fractured" LUTs that share some inputs. We represent modes in the architecture definition in much the same way as how hierarchy is expressed. Each mode is represented as a subcluster where the subcluster can only be used if none of its siblings (in the logic block hierarchy) are used.

Fig. 2 shows an example of an architecture with soft logic blocks and I/O logic blocks. The soft logic block consists of a subcluster of two 3-input LUT primitives and a single 4-to-1 mux primitive. Unlike the routing muxes within the FPGA that implement nets, this mux primitive is used to implement actual logic in the user netlist. The data lines of the mux are driven by the input pins of the logic block. The two 3-LUTs drive the select lines of the mux primitive. The LUT primitives have special properties. They can operate as interconnect wires in addition to logic. The two 3-LUTs are driven by four logic block inputs so they share two input pins. Unlike LUT and mux primitives, input and output pads are implemented in their own dedicated logic blocks. An I/O logic block has two modes of operation. It can implement one input pad or one output pad.

3.1 Packing Problem

The packing problem is defined as the assignment of netlist atoms to primitives, while optimizing for multiple objectives, under the constraint that the final mapping is *legal*. Ultimately, we want the packer to produce a netlist that gives good area and delay results after placement and routing. To acheive this practically, we set the objectives of packing to minimize the number of logic blocks, minimize the number of connections that must route across multiple logic blocks, and group together connected atoms where those connections are timing critical. A legal solution is a packing solution that can be physically realized. We determine legality by checking for various conditions. The main conditions are as follows: 1) All atoms are assigned to unique primitives and each of those primitives can implement the atom assigned to it, 2) All nets within a logic block can be routed, and 3) Modes within a logic block are mutually exclusive.

3.2 Example of Packing

Fig. 3 gives an example packing of the input circuit shown in Fig. 1 into an FPGA with the logic block types shown in Fig. 2. The lightly shaded primitives, such as LUT:L1 and MUX:M1, name the atoms that they implement. The dark LUT shows a LUT that is used to implement a wire. The thick edges show the physical interconnect edges that implement nets. The I/O logic blocks are set to either inpad or outpad mode, depending on what I/O atom got assigned to them.

4. INTERCONNECT-AWARE PACKING

This section describes the modifications that we made to an existing packing tool (AAPack) [9] to enable the packer to adapt to the underlying interconnect of the target logic blocks. We begin with an overview of the original packing algorithm. We then describe the various modifications that we made.

The packing algorithm starts by selecting an empty logic block in the FPGA to be packed. The algorithm then fills the logic block by attempting to assign candidate atoms to unused primitives. Candidate atoms are chosen using an attraction function, where atoms with a higher score are considered before atoms with a lower score. Once the logic block is full, the algorithm "closes" that block and opens up a new, empty, logic block to be filled. Packing terminates when all atoms in the netlist have been mapped into logic blocks.

The attraction function is a weighted sum of two terms [17]. The first term scores the timing criticality of a candidate while the second term scores the connectivity of a candidate with the logic block that is currently being filled. The algorithm first only considers unpacked atoms that share one or more nets with the packed atoms inside the currently active logic block. This keeps the algorithm scalable and prevents unrelated logic from being packed together. If no such atoms remain, then the algorithm selects atoms from unrelated logic.

Up to this point, the algorithm description is about the same as many other greedy FPGA packers [12] [3] [4]. The part that distinguishes this packer, the part that enables architecture adaptiveness, is in the stage where the algorithm determines which primitive, if any, a candidate netlist atom should map to within a logic block. We call this stage intra-logic block placement and routing.

Intra-logic block placement finds a suitable primitive within the logic block to assign the candidate atom then assigns modes within the logic block based on that placement. This stage is also responsible for checking most of the legality constraints. These checks include whether or not the primitive can implement the atom, if modes assigned are legal, and if basic routability (in the form of pin counting) passes. If placement is successful, then the original packing algorithm [9] would invoke intra-logic block routing to attempt detailed routing, using the PathFinder algorithm [13], to ensure routability. If detailed routing fails, then the placement and routing process repeats itself until a successful primitive is found or until there are no more unused primitives to try.

Among all the legality checks, detailed routing consumes, by far, the most time. The main focus of our techniques is to avoid this computationally intensive check when the interconnect in the architecture is simple enough for us to skip it.

4.1 Speculative Packing

Speculative packing is a technique to avoid unnecessary invocations of detailed routing. This technique first attempts to optimistically pack a logic block by not invoking detailed routing until the logic block is filled. We call the optimistically filled logic block the *speculated solution*. If detailed routing of the speculated solution succeeds, then the solution is accepted. Otherwise, the packer rejects the speculated solution and reverts back to the conservative method of [9] that invokes detailed routing for every partial packing.

The runtime impact from speculative packing depends heavily on how often the final route of a speculated solution succeeds. In the best case, the final route always succeeds resulting in speedup. In the worst case, the final route never succeeds which results in wasted speculation time. Thus, if a logic block contains simple interconnect from which the packer can form routable speculated solutions, then speculative packing enables the packer to expend less computational effort routing. If a logic block contains more complex interconnect, then the computational effort expended by the packer depends on how often the packer assembles a routable speculated solution.

4.2 Interconnect-Aware Pin Counting

Pin counting is a technique that approximates the routability problem with a simpler counting problem. Pin counting checks if a particular assignment of atoms to a logic block/-subcluster uses more pins than supplied by the logic block-/subcluster. If pins are overused, then that assignment is proven unroutable. If pins are not overused, then in the pin counting approach, we optimistically assume that the assignment is routable. Pin counting is one of the many checks performed in intra-logic block placement. This implies that during speculative packing, when detailed routing is skipped, pin counting becomes the only check for routability. Therefore, more accurate pin counting reduces computational effort by increasing the chance that speculated solutions will route.

Interconnect-aware pin counting is our more precise version of pin counting. In addition to analyzing pins, this technique also analyzes the underlying physical interconnect with the intention of capturing clues about how those pins are related. We begin by describing what information this technique extracts from the interconnect, and then we describe how packing uses that information.

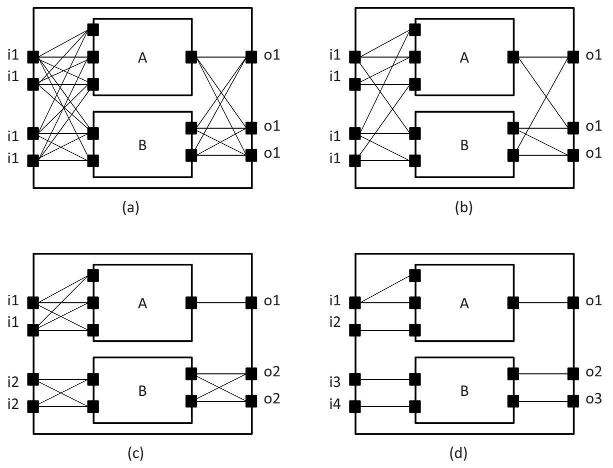

Figure 4: Examples on how pins are grouped into pin classes

Prior to the packing stage, we analyze the architecture of each logic block, and group the pins of each block and subcluster into separate *pin classes* based on the interconnect structures. Intuitively, pin classes are an attempt to approximate arbitrary interconnect with a set of non-overlapping full crossbars. Input pins of the same class drive the same crossbar, output pins of the same class are driven by the same crossbar. Pins are grouped into pin classes using the following process: First, all pins are set to have their own, individual, pin class. Then, pin classes are merged together based on connectivity while subject to constraints. Pin classes are constrained by subcluster/logic block and by type. Pins in the same pin class must be on the periphery of the same subcluster/logic block. Furthermore, pins of the same pin class must be either all input pins or all output pins. Subject to these constraints, if two pins of a subcluster/logic block belong to different pin classes but can connect (through the interconnect) to a common primitive pin within that subcluster/logic block, then those pin classes are merged together. When two pin classes merge, all pins contained in either of the original pin classes are grouped into one new pin class. In the event that the primitive has logically equivalent pins (for example, an AND gate has logically equivalent input pins), then those primitive pins are considered as one pin for the purposes of determining pin classes. At any point during the construction of pin classes, if a pin gets assigned to two different pin classes, then all pins in both pin classes are merged into a single pin class. The process ends when no more pin classes can be merged together.

Fig. 4 illustrates different examples of pin classes on a subcluster with two primitives, four input pins, and three output pins. The labels on the subcluster pins show which pin class each pin belongs to. Fig. 4 (a) has a large, well populated crossbar at the inputs and outputs. The subcluster input pins all belong to the same pin class *i1* and the subcluster output pins all belong to another pin class *o1.* Fig. 4 (b) has a sparser crossbar than (a). Our technique optimistically approximates these cases as the same, thus (b) has the

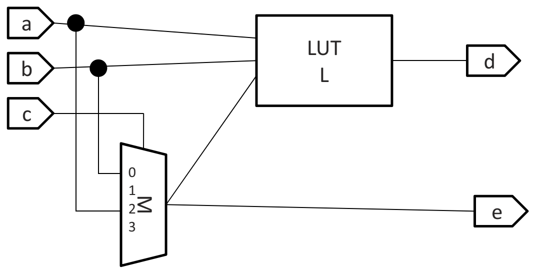

Figure 5: Example netlist to illustrate pin counting.

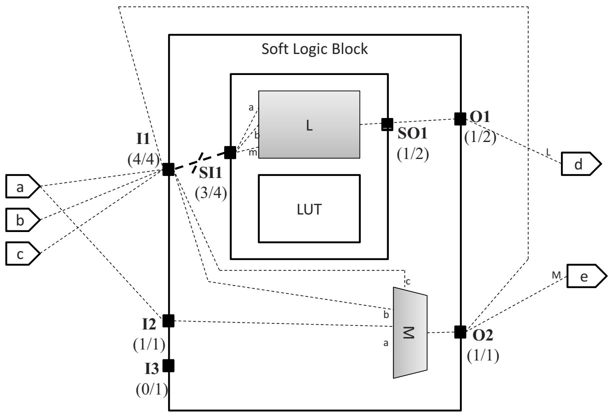

Figure 6: Example intermediate solution to illustrate pin counting.

same pin classes as (a). Fig. 4 (c) has disconnected smaller crossbars. This is reflected in the two pin classes for the inputs and two pin classes for the outputs. Finally, Fig. 4 (d) has no interconnect flexibility so all subcluster pins belong to separate pin classes.

During packing, every time a candidate atom is placed inside a logic block, pin counting updates the utilization of the pin classes of used subclusters within the logic block then updates the pin classes of the logic block itself. If, after the update, there exists a pin class that uses more pins than is supplied by that pin class, then pin counting declares the intermediate solution unroutable. Without loss of generality, we describe the update procedure for just the logic block. A net adds a count of one to a pin class of input pins if and only if the net drives a primitive input pin through that pin class and the driver of that net cannot reach the primitive input pin solely from within the logic block. A net adds a count of one to a pin class of output pins if and only if the primitive output pin of that net drives that pin class and there exists one primitive input pin driven by the net that cannot be reached solely from within the logic block.

To illustrate the nuances of pin counting, we revisit the logic block described in Fig. 2. We show the effect of packing the netlist in Fig. 5 to that logic block. This netlist consists of a LUT L and a logical mux M. Fig. 6 shows an intermediate packing solution that placed the LUT L in the top LUT position and the mux M in the mux location. We start by describing the pin classes in the logic block, then we describe the utilization of each pin class.

The logic block input pin classes are I1, I2, and I3. The logic block output pin classes are O1 and O2. The dual-LUT subcluster input pin class is SI1 and the subcluster output pin class is SO1. There is some subtlety in determining pin class I1. All four top input pins of the logic block belong to the same pin class. A LUT has logically equivalent inputs so the top three input pins are grouped together and the second, third, and fourth input pins are grouped together. Moreover, since the second and third input pins are common to both groups, all four pins are merged into the same pin class. The capacity of each pin class is determined by the number of pins it grouped. We label this value in the figure as the denominator of the fraction displayed beside each pin class. For pin class I1, the capacity is 4.

The utilization of each pin class is determined by the nets connected to the primitives within the logic block. This value is displayed as the numerator in the fraction beside each pin class in the figure. Observe the following sublety: Net a requires two logic block input pins because of the lack of internal flexibility in the logic block. This behaviour is captured by the separation into pin classes I1 and I2. This is in contrast to net b which only needs to consume one pin because of internal fanout within the logic block. This connectivity is captured in pin class I1. Net M from the logical mux must traverse outside the logic block to reach the LUT input. This is represented as consuming one count of pin class O2 and one count of pin class I1. Net c illustrates how interconnect-aware pin counting is optimistic. Without the ability to detect that it is necessary to route through the dual-LUT subcluster to reach the mux select line, we see that our pin counting technique optimistically uses 3 of 4 pins in SI1, when in fact all 4 must be used in a detailed route. These examples illustrate which properties interconnect-aware pin can capture and which it cannot capture.

To summarize, we list the properties and limitations of interconnect-aware pin classes as follows:

- Acts as an optimistic filter. Cases that fail interconnect-aware pin counting will fail to route while cases that pass may or may not successfully route.
- Sparse interconnect is approximated as fully flexible.
- Does not account for situations where a net routes through a subcluster without connecting to any primitive within the subcluster.
- Internal feedback/feedfoward connections within a logic block/subcluster are discovered before packing and accounted for during pin counting.
- Only returns pass/fail. Does not give hints to guide future candidate selection.

4.3 Pre-Packing

Logic blocks sometimes contain inflexible routing structures. These structures can cause complications in a greedy packer because different stages of the packer become necessarily coupled. We illustrate this coupling using carry chains as an example. A carry chain is an important structure that enables the fast computation of wide logical adders by chaining together smaller physical adders using fast, inflexible carry links. In the packing stage of the VTR CAD flow, a logical adder is represented by multiple smaller adder

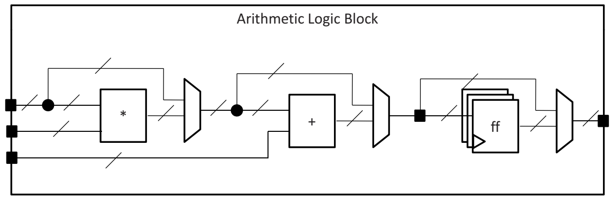

Figure 7: A bus-based arithmetic logic block

atoms that link together to form the logical adder. The packer must map the adder atoms to physical adder primitives in such a way that the physical chain can implement those logical links. An incorrect grouping or placement of the atoms during packing can result in failed (internal-to-the-block) routing because carry connections may become impossible to route. This example shows how inflexibility in interconnect can cause strong coupling among the candidate selection and placement stages in packing. This coupling is not unique to carry chains. We observe this coupling effect in multiple other logic block constructs including primitives with registered inputs/outputs, datapath arithmetic blocks with compound operations such as multply-add, and others.

We employ a pre-packing technique to capture coupling from restrictive interconnect in a generic and simple way. The architect is asked to identify (in the architecture file) groups of primitives joined together using inflexible interconnect. These groups and their links are called *pack patterns*. Before packing, groups of netlist atoms that match a pack pattern are grouped together into what is called a *molecule*. We call this stage the *pre-packing* stage. During packing, molecules are treated as though they are an atom and can only map to primitives that form the same pack pattern as the molecule.

Fig. 7 shows an example of the concept of pack patterns and molecules. This arithmetic logic block can perform both basic multiplication and addition, as well as combined operations such as multiply-add and registered arithmetic. If the architect intends for combined operations to be kept together during packing, then the architect should indicate that intent by specifying four pack patterns as follows: 1) Multiply-add, 2) Registered multiply, 3) Registered add, and 4) Registered multiply-add.

4.4 Other Modifications

Unlike the previous modifications which enable the packer to adapt to the detailed interconnect of a logic block, the enhancements described in this section are general improvements to the packer. We discuss three important improvements: First, more accurate timing analysis during packing that uses delay values from the architecture description file. Second, best-fit intra-logic block placement. Third, special case handling for high fanout nets.

Accurate Timing Analysis

The original AAPack tool computes timing criticality based on a delay graph that is a function of only the netlist. We modified the delay graph to also include architectural information. The delay model of an atom is taken from the delay model of the smallest physical primitive in the architecture that can implement that atom. In addition, we model interconnect delay between atoms with a single constant based on inter-logic block wire delays.

Best-Fit Placement

We modified the intra-logic block placement function to employ best-fit placement instead of first-fit placement. The intra-logic block placer in [9] employs a first-fit algorithm to determine where to place a candidate atom within a logic block. This can lead to quality of results being heavily dependant on how a logic block is described in the architecture file because the placer will not examine any other primitives after it encounters one unused primitive that can implement the candidate atom.

Our best-fit placement iterates through all valid primitives and returns the primitive with the least cost that can implement the candidate atom. The base cost of a primitive is equal to the number of pins of that primitive. This encourages the placer to select smaller primitives before bigger primitives. When a primitive is used, the cost of each unused primitive is reduced by 0.1^a where a is the depth of the used primitive to the closest ancestor of that unused primitive and the value 0.1 is an empirically derived parameter. This encourages the placer to consider primitives in used sections of the logic block hierarchy before unused sections.

High Fanout Nets Handling

The original AAPack tool, along with many prior academic packers, have scalability problems with circuits that contain high fanout nets. In the original AAPack, all atoms connected to a high fanout net are considered during packing. Since a high fanout net reaches many atoms, that pool of weakly connected atoms is considered several times over the course of packing. This ultimately results in a runtime cost that is quadratic with the number of terminals for a high fanout net.

We modified the attraction function to initially ignore high fanout nets when packing to a logic block. When these candidate atoms are exhausted, then the algorithm prioritizes selecting candidate atoms connected by high fanout nets that are connected to the currently active cluster before considering completely unrelated logic. A net is considered to be high fanout when it exceeds 64 terminals.

5. RESULTS

In this section, we measure the impact of the new algorithms introduced in the previous section on the quality of results and runtime of the packer. We also illustrate how the new, complete packer adapts to architectures with increasing levels of interconnect complexity. We label the prior version of AAPack, released in the VTR 1.0 suite, as AAPack 6.0, and this new version released in VTR 7.0 as AAPack 7.0.

5.1 Experimental Setup

We use the VTR 7.0 CAD flow [11] in these experiments as shown in Fig. 8. This flow takes as input a benchmark Verilog circuit and an FPGA architecture description file. The flow maps the circuit to the architecture described in that file then outputs statistics about that final mapping. We use Odin II for elaboration, ABC for logic synthesis, one of AAPack 6.0 or AAPack 7.0 for packing, and VPR 7.0 for placement and routing. VPR is left at default values [10] except the placement option inner_num is set to 10.0[1].

[1]This placement algorithm option aligns VPR 7 placement with prior versions of VPR.

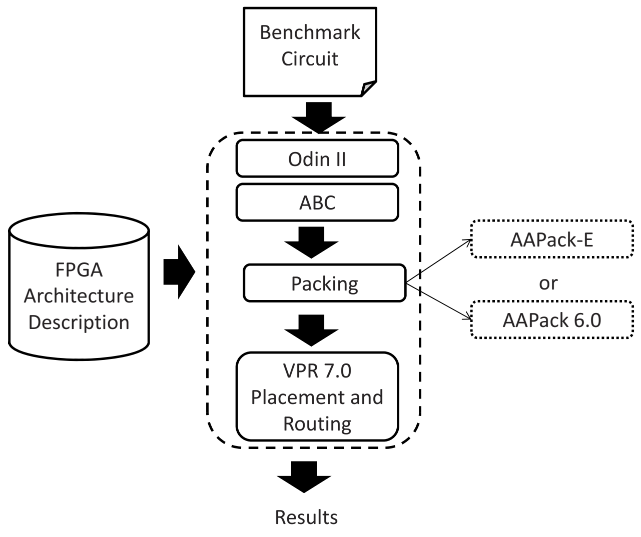

Figure 8: The experimental CAD flow

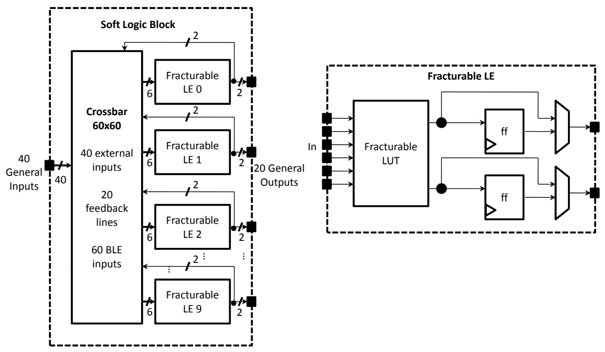

Figure 9: The baseline soft logic block architecture.

We use the VTR 7.0 benchmarks for our experiments. The VTR 7.0 benchmarks are a standard set of Verilog circuits that come from a variety of different applications including computer vision, medical, math, soft processors, etc. These circuits contain heterogeneous elements, such as memories and multipliers, which differentiate them from older FPGA benchmarks. These benchmarks range from a few hundred 6-LUTs in size to just over a hundred thousand 6-LUTs. Most contain memories and/or multipliers of varying quantities and sizes. For example, the circuit `stereovision2` contains 564 logical multipliers and the circuit `mcml` contains 30 logical multipliers and 10 memory blocks totalling 5 Mb. These benchmarks are very similar to the VTR 1.0 benchmarks [17], but have some minor Verilog changes.

The baseline FPGA architecture used in these experiments is a 40nm CMOS heterogeneous architecture released in VTR 7.0 called `k6_frac_N10_mem32K_40nm.xml`. Fig. 9 shows the soft logic blocks of this architecture which is loosely modelled from an Altera Stratix IV FPGA [1]. A soft logic block has 40 inputs, 20 outputs, and 10 fracturable 6-LUTs. A fully populated internal crossbar connects all logic block inputs to fracturable LUT inputs and provides feedback connections within the logic block. Each fracturable 6-LUTs can optionally operate as two five LUTs with some shared inputs. The 5-LUTs are set to share all 5 inputs. This sharing more closely resembles a Virtex 6 fracturable LUT [21] than a Stratix IV fracturable LUT. It was chosen because this reduces the size of the internal crossbar to more closely match the number of switch points in a Stratix IV depopulated internal crossbar. All LUTs have optionally registered outputs. This architecture contains fracturable multipliers, where each multiplier can operate as one large 36x36 multiplier or two fracturable 18x18 multipliers. A fracturable 18x18 multiplier can operate as one 18x18 multipier or two 9x9 multipliers. Finally, this architecture contains configurable memories. Each memory has 32Kb and can operate in aspect ratios ranging from 32Kx1 to 512x64. The area and delay values of this architecture are mostly chosen to match a Stratix IV FPGA [1].

The machine used in this experiment has two Intel Xeon 5160 processors running at 3 GHz. Each processor has two cores with 4 MB of L2 cache. Each machine has a total of 8 GB of shared memory. Although this machine is capable of parallelism, we chose to run our experiments single-threaded.

Table 1: Results of VTR CAD flow using AAPack 7.0 across 19 benchmarks.

Circuit	Min W	Crit Delay (ns)	Pack Time (s)	Num Ex Nets	Num CLBs
bgm	114	25.71	198.2	21.1K	2930
blob_merge	72	10.47	17.45	3069	543
boundtop	58	6.51	6.56	2200	233
ch_intrinsics	48	3.89	0.77	430	37
diffeq1	50	21.50	0.72	717	36
diffeq2	52	17.04	0.52	468	27
LU8PEEng	108	111.81	84.89	16.3K	2104
LU32PEEng	168	115.39	429.25	54.2K	7128
mcml	94	81.73	681.78	52.4K	6615
mkDelayWorker32B	80	7.40	19.99	5224	447
mkPktMerge	48	4.57	0.64	972	15
mkSMAdapter4B	54	5.85	5.26	1597	165
or1200	72	13.29	8.3	2499	257
raygentop	70	5.04	7.54	1964	173
sha	50	13.77	10.58	1304	209
stereovision0	58	4.23	33.54	7936	905
stereovision1	102	5.65	38.63	11.1K	889
stereovision2	154	19.68	82.37	34.5K	2395
stereovision3	34	2.72	0.17	122	13

5.2 AAPack 7.0 vs AAPack 6.0

In this experiment, we compare AAPack 7.0 with the original AAPack 6.0 in the context of the full CAD flow. The architecture used is the baseline architecture previously described.

Table 1 shows the absolute values of running the VTR flow using AAPack 7.0. The leftmost column lists the circuit used. After that, from left to right, the columns are as follows: 1) The minimum channel width (min W) needed to route the circuit; 2) The critical path delay in nanoseconds when the circuit is routed at 1.3 times minimum W for the current flow (this follows historical precedant to route the circuit under reasonable stress); 3) The time needed to pack the circuit in seconds; 4) The number of external nets (nets that are routed between logic blocks); and 5) The number of soft logic blocks used in each benchmark. This table serves as the baseline values from which the later relative comparisons are made.

Table 2: Relative comparison of AAPack 7.0 over AAPack 6.0.

Circuit	Min W	Crit Delay	Pack Time	Num Ex Nets	Num CLBs
bgm	0.71	0.83	0.45	0.89	1.00
blob_merge	0.78	1.00	0.16	0.82	1.00
boundtop	0.78	0.85	0.03	0.86	0.95
ch_intrinsics	0.89	1.01	0.04	0.92	0.93
diffeq1	1.00	0.96	0.10	0.94	0.92
diffeq2	1.08	0.93	0.07	0.91	1.00
LU8PEEng	0.77	0.95	0.18	0.98	1.02
LU32PEEng	0.72	0.97	0.18	0.98	1.02
mcml	0.48	0.97	0.08	0.89	0.97
mkDelayWorker32B	0.80	0.92	0.07	0.91	1.01
mkPktMerge	1.00	1.06	0.14	0.99	1.00
mkSMAdapter4B	0.77	0.89	0.02	0.91	0.97
or1200	0.86	0.98	0.06	0.87	0.97
raygentop	0.92	1.00	0.11	0.90	1.01
sha	0.74	1.00	0.04	0.71	0.99
stereovision0	0.66	0.78	0.06	0.92	1.09
stereovision1	0.82	0.92	0.06	0.92	1.09
stereovision2	0.71	0.92	0.08	0.94	1.06
stereovision3	1.00	0.90	0.05	0.89	1.00
geomean	0.80	0.94	0.08	0.90	1.00
stdev	0.17	0.07	0.12	0.05	0.04

Table 2 measures how well AAPack 7.0 performs relative to AAPack 6.0. Each value is presented as a ratio of AAPack 7.0 over AAPack 6.0. The columns are the same as columns 1-5 of Table 1. For packer runtime, AAPack 7.0 is 12-fold faster than AAPack 6.0. This illustrates the effectiveness of our techniques. The interconnect of this architecture is simple enough that the combination of interconnect-aware pin counting and LUT/FF molecules is sufficient for speculated solutions to always route. This enables AAPack 7.0 to successfully skip many of the intermediate detailed routing checks that AAPack 6.0 invokes resulting in a speed up. Profiling the packer revealed that detailed routing checks dropped from over 90% of pack time in AAPack 6.0 to approximately 10% in AAPack 7.0 which further confirms this causative link. In terms of quality of results, AAPack 7.0 absorbs nets better resulting in a 10% reduction in the number of external nets which leads to the 20% reduction in minimum channel width. Critical path delay is reduced by 6%. These quality of results improvements show that pre-packing, best-cost placement, and more accurate timing analysis are effective techniques towards reducing inter-logic block routing stress and enabling better delay optimizations. We conclude that AAPack 7.0 produces packed circuits that are better than AAPack 6.0 and can do so at much lower runtime.

Table 3 measures the effect of packing quality on placement and routing runtime. The leftmost column lists the circuit used. The middle column lists the relative time needed to place the circuit. The rightmost column lists the relative time needed to route the circuit at a fixed route of 1.3 times minimum W. Each value is presented as a ratio of AAPack 7.0 over AAPack 6.0. The results show that placement time is reduced by 8% and fixed route time by 18% on average. The speedup from using AAPack 7.0 in placement and fixed channel width routing is a result of the reduction in the number of external nets. Fewer external nets reduces the time needed for the placer to update costs and reduces the load on the router. We conclude that the higher quality packing from AAPack 7.0 reduces the runtime of placement and routing.

Table 3: Relative place-and-route runtime comparison of circuits packed by AAPack 7.0 over circuits packed by AAPack 6.0.

Circuit	Place Time	Fixed Route Time
bgm	0.90	0.68
blob_merge	0.89	0.74
boundtop	0.88	0.65
ch_intrinsics	0.87	0.83
diffeq1	0.93	0.91
diffeq2	0.92	0.92
LU8PEEng	0.89	0.78
LU32PEEng	0.96	0.93
mcml	0.96	0.59
mkDelayWorker32B	0.89	0.87
mkPktMerge	1.01	0.95
mkSMAdapter4B	0.90	0.72
or1200	0.90	0.78
raygentop	0.94	0.85
sha	0.77	0.80
stereovision0	0.92	0.82
stereovision1	0.95	1.04
stereovision2	1.01	0.76
stereovision3	1.09	1.25
geomean	0.92	0.82
stdev	0.04	0.13

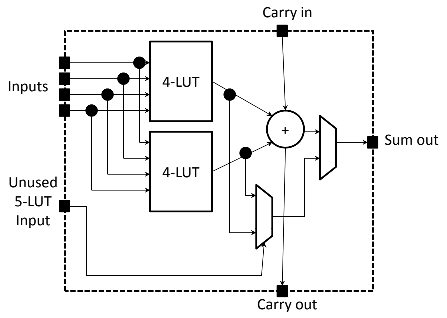

Figure 10: Interaction between 5-LUT and carry chain adder

5.3 Architecture Adaptiveness

In this experiment, we measure how well AAPack 7.0 adapts to architectures with varying levels of interconnect complexity. We make two main comparisons. The first compares how AAPack 7.0 performs on different architectures with respect to a baseline architecture. The second compares how AAPack 7.0 performs on these same architectures against AAPack 6.0.

We run the VTR benchmarks using the same VTR flow as earlier on five different architectures with varying types of interconnect. All these architectures are variations of the `k6_frac_N10_mem32K_40nm.xml` baseline. The first architecture has simpler soft logic blocks by replacing the fracturable 6-LUTs with non-fracturable 6-LUTs. The second architecture adds carry chains to the baseline. Fig. 10 shows how the adder is integrated with the fracturable LUT. When the fracturable 6-LUT is operating in dual 5-LUT mode, each 5-LUT further fractures into two 4-LUTs that drive one hardened adder bit. Dedicated carry links join all 20 hard adders together and also establishes connections to the

Table 4: Comparison of how AAPack 7.0 performs on different architectures vs the baseline architecture.

Architecture	Pack Time	Num Ext Nets	Num CLBs
Non-fracturable	0.41	1.02	1.33
Carry Chain	1.50	1.20	1.14
Xbar 0.5	1.73	1.02	1.01
Xbar 0.25	1.46	1.09	1.01
Xbar 0.1	14.21	1.09	1.07

soft logic block carry input and carry output pins. The next three architectures replace the complete internal crossbar of the baseline with a depopulated crossbar. A depopulated crossbar is a crossbar where some of the switch points are removed thus reducing area at the cost of less connectivity. A crossbar that uses 25% of all possible switch points is 25% populated. The third, fourth, and fifth architectures have crossbars populated at 50%, 25%, and 10% respectively.

Table 4 shows the results of AAPack 7.0 on different architectures normalized to the results from the baseline architecture. The leftmost column lists the architecture being investigated. Moving rightwards, the next columns are pack time, number of external nets, and number of soft logic blocks. The values shown are the geometric mean across all 19 benchmarks normalized to the baseline. These results show the general trend that AAPack 7.0 runtime is faster for architectures with simpler interconnect and slower for architectures with more complex interconnect. AAPack 7.0 runs more than twice as fast for a simple, non-fracturable LUT architecture. AAPack 7.0 runs slower for architectures with a carry chain or a depopulated internal crossbar. At the extreme, a very sparse 10% populated crossbar runs 14-fold slower than the same architecture with a full crossbar because interconnect-aware pin counting no longer accurately captures the complexities of sparsity. This experiment also demonstrates other findings. First, it shows a proof-of-concept that pre-packing enables the packer to target carry chain architectures. Second, this experiment shows that quality of results for architectures with depopulated internal crossbars can remain fairly high. For example, at 50% population, the packer produces 1% more soft logic blocks and 2% more external nets on average.

The next experiment examines how AAPack 7.0 performs compared to AAPack 6.0 for the same set of architectures. We exclude the carry chain architecture in this comparison because AAPack 6.0 is not capable of packing to architectures with carry chains. Table 5 shows the results of this experiment. The columns are the same as for Table 4. The values shown are the geometric mean across all 19 benchmarks of the AAPack 7.0 runs normalized to the AAPack 6.0 runs. For classic, non-fracturable LUT, architectures, pack time is 6.7-fold faster with 3% fewer external nets. AAPack 7.0 is 6-fold faster for soft logic blocks that contain crossbars at 50% and 25% population. We notice a large 10% to 12% reduction in external nets which indicates that AAPack 7.0 packs to soft logic blocks with depopulated crossbars better than AAPack 6.0. Lastly, we notice that for the very low 10%-populated crossbar, AAPack 6.0 no longer packs efficiently requiring 27% more CLBs than AAPack 7.0. However, AAPack 7.0 only runs 2.4-fold faster than AAPack 6.0 for this architecture because the sparsity of the crossbar causes AAPack 7.0 to invoke detailed routing for many of the partially packed solutions. These results show that AAPack 7.0 performs better than AAPack 6.0 across all architectures and AAPack 7.0 better adapts to architectures with complex interconnect than AAPack 6.0.

Table 5: Comparison of AAPack 7.0 vs AAPack 6.0 across different architectures.

Architecture	Pack Time	Num Ext Nets	Num CLBs
Non-fracturable	0.15	0.97	1.00
Xbar 0.5	0.15	0.90	1.00
Xbar 0.25	0.13	0.88	1.00
Xbar 0.1	0.41	0.87	0.79

6. CONCLUSIONS AND FUTURE WORK

We have presented key enhancements in AAPack 7.0 that enables the tool to adapt to the underlying interconnect of an FPGA architecture. These enhancements speed up a state-of-the-art flexible packer by 12-fold, while simultaneously lowering minimum channel width by 20% and lowering critical path delay by 6% on a modern style FPGA. We demonstrate that these enhancements provide more robust packing across a diverse range of architectures, including architectures with carry chains and architectures with depopulated crossbars. This work is a key step towards a flexible packing tool that can adapt its computational effort based on the difficulty of the underlying interconnect architecture.

In the future, we intend to use AAPack 7.0 to investigate new logic block architectures. We identify the interplay between fracturable LUTs, carry chain architecture, and depopulated crossbars as an interesting direction to explore.

7. REFERENCES

[1] Altera Corporation. Stratix IV Device Family Overview. `http://www.altera.com/literature/hb/stratix-iv/stx4_siv51001.pdf`, November 2009.

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

[3] E. Bozorgzadeh, S. Memik, and M. Sarrafzadeh. RPack: Routability-driven Packing for Cluster-Based FPGAs. In *ASP-DAC '01: Proceedings of the 2001 Asia and South Pacific Design Automation Conf.*, pages 629–634, New York, NY, USA, 2001. ACM.

[4] D. Chen, K. Vorwerk, and A. Kennings. Improving Timing-Driven FPGA Packing with Physical Information. *Int'l Conf. on Field Programmable Logic and Applications*, pages 117–123, 2007.

[5] J. Cong, J. Peck, and Y. Ding. RASP: A general logic synthesis system for SRAM-based FPGAs. In *Proceedings of the 1996 ACM fourth international symposium on Field-programmable gate arrays*, pages 137–143. ACM, 1996.

[6] G. Lemieux and D. Lewis. *Design of Interconnection Networks for Programmable Logic.* Kluwer Academic Publishers, Norwell, Massachusetts, 2004.

[7] J. Lin, D. Chen, and J. Cong. Optimal Simultaneous Mapping and Clustering for FPGA Delay Optimization. In *ACM/IEEE Design Automation Conf.*, pages 472–477, 2006.

[8] A. Ling, J. Zhu, and S. Brown. Scalable Synthesis and Clustering Techniques Using Decision Diagrams. *IEEE Trans. on CAD*, 27(3):423, 2008.

[9] J. Luu, J. Anderson, and J. Rose. Architecture Description and Packing for Logic Blocks with Hierarchy, Modes and Complex Interconnect. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, FPGA '11, pages 227–236, New York, NY, USA, 2011. ACM.

[10] J. Luu, J. Goeders, T. Liu, A. Marquardt, I. Kuon, J. Anderson, J. Rose, and V. Betz. VPR User's Manual (Version 7.0). `http://code.google.com/p/vtr-verilog-to-routing/downloads/list`, 2013.

[11] J. Luu, J. Goeders, M. Wainberg, A. Somerville, T. Yu, K. Nasartschuk, M. Nasr, S. Wang, T. Liu, N. Ahmed, K. B. Kent, J. Anderson, J. Rose, and V. Betz. Verilog-to-Routing 7.0. `https://code.google.com/p/vtr-verilog-to-routing/`, 2013.

[12] A. Marquardt, V. Betz, and J. Rose. Using Cluster-Based Logic Blocks and Timing-Driven Packing to Improve FPGA Speed and Density. *ACM Int'l Symp. on FPGAs*, pages 37–46, 1999.

[13] L. McMurchie and C. Ebeling. PathFinder: A Negotiation-Based Performance-Driven Router for FPGAs. In *ACM Int'l Symp. on FPGAs*, pages 111–117, 1995.

[14] K. E. Murray, S. Whitty, S. Liu, J. Luu, and V. Betz. Titan: Eabling Large and Complex Benchmarks in Academic CAD. 2013.

[15] G. Ni, J. Tong, and J. Lai. A New FPGA Packing Algorithm Based on the Modeling Method for Logic Block. In *IEEE Int'l Conf. on ASICs*, volume 2, pages 877–880, Oct. 2005.

[16] D. Paladino. Academic Clustering and Placement Tools for Modern Field-Programmable Gate Array Architectures. Master's thesis, University of Toronto, Toronto, Ontario, Canada, 2008.

[17] J. Rose, J. Luu, C. W. Yu, O. Densmore, J. Goeders, A. Somerville, K. B. Kent, P. Jamieson, and J. Anderson. The VTR Project: Architecture and CAD for FPGAs from Verilog to Routing. In *ACM Int'l Symp. on FPGAs*, pages 77–86, 2012.

[18] A. Sharma, S. Hauck, and C. Ebeling. Architecture-adaptive routability-driven placement for FPGAs. In *Field Programmable Logic and Applications, 2005. International Conference on*, pages 427–432. IEEE, 2005.

[19] A. Singh, G. Parthasarathy, and M. Marek-Sadowksa. Efficient Circuit Clustering for Area and Power Reduction in FPGAs. *ACM Trans. on Design Automation of Electronic Systems*, 7(4):643–663, Nov 2002.

[20] K. Wang, M. Yang, L. Wang, X. Zhou, and J. Tong. A Novel Packing Algorithm for Sparse Crossbar FPGA Architectures. In *Int'l Conf. on Solid-State and Integrated-Circuit Technology*, pages 2345–2348, 2008.

[21] Xilinx Inc. Xilinx Virtex-6 Family Overview. `http://www.xilinx.com/support/documentation/data_sheets/ds150.pdf`, 2009.