Architecture Description and Packing for Logic Blocks with Hierarchy, Modes and Complex Interconnect

Jason Luu, Jason Anderson, and Jonathan Rose
The Edward S. Rogers Sr. Department of Electrical and Computer Engineering
University of Toronto, Toronto, ON, Canada
jluu|janders|jayar@eecg.utoronto.ca

ABSTRACT

The development of future FPGA fabrics with more sophisticated and complex logic blocks requires a new CAD flow that permits the expression of that complexity and the ability to synthesize to it. In this paper, we present a new logic block description language that can depict complex intra-block interconnect, hierarchy and modes of operation. These features are necessary to support modern and future FPGA complex soft logic blocks, memory and hard blocks. The key part of the CAD flow associated with this complexity is the packer, which takes the logical atomic pieces of the complex blocks and groups them into whole physical entities. We present an area-driven generic packing tool that can pack the logical atoms into any heterogeneous FPGA described in the new language, including many different kinds of soft and hard logic blocks. We gauge its area quality by comparing the results achieved with a lower bound on the number of blocks required, and then illustrate its explorative capability in two ways: on fracturable LUT soft logic architectures, and on hard block memory architectures. The new infrastructure attaches to a flow that begins with a Verilog front-end, permitting the use of benchmarks that are significantly larger than the usual ones, and can target heterogenous FPGAs.

Categories and Subject Descriptors

B.6.3 [**Design Aids**]: Hardware description languages, Optimization

General Terms

Algorithms, Design, Languages, Measurement, Performance

1. INTRODUCTION

As the semiconductor industry evolves and accounts for the prohibitive cost of custom chip design and fabrication, together with the continued exponential growth in logic capacity per die, there is a need to make pre-fabricated and programmable chips more capable. That enhanced capability may in part be expressed through more complex programmable logic blocks. These logic blocks may perform certain computations more efficiently, or store data, or perhaps contain novel soft logic structures.

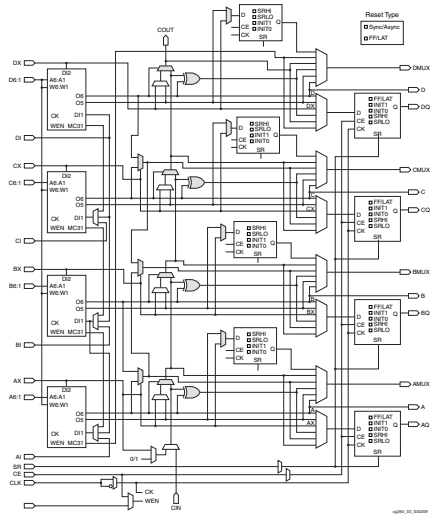

Figure 1: Commercial Virtex-6 logic block

Today's publicly available FPGA CAD tools lack the ability to target the complexity present in modern commercial architectures - the Altera Stratix IV [1] and Xilinx Virtex 6 [2] FPGAs contain highly complex soft logic blocks, hard memories and mulitpliers. For example, it simply isn't possible to represent the details of the Virtex-6 logic block, illustrated in Fig. 1 [3], in the architecture description language provided in VPR 5.0 [20]. While commercial tools can be used to synthesize to this exact architecture, there is no capability for researchers to explore new issues (such as process variability) for that device, or to modify important aspects of the architecture.

Furthermore, the MCNC benchmark circuits [33] often used in research are no longer representative of modern or future FPGA applications because they are no larger than a few thousand 4-LUTs while commercial FPGAs today can target applications that contain hundreds of thousands and soon millions or more 4-LUT-equivalents. There is a need, then, for modern, public benchmarks and CAD tools that can deal with the heterogeneity inherent in those benchmarks in order to do relevant, scientific research on FPGA architecture and CAD.

In this paper, we describe two key new capabilities for an FPGA CAD flow that provide the ability to describe and synthesize for the necessary complexity: first, we present a new logic block description language that can express far more complex logic blocks than is currently possible with

FPGA'11, February 27–March 1, 2011, Monterey, California, USA.

any publicly available toolset. It can describe complex logic blocks with arbitrary internal routing structures (such as all the small muxes in Fig. 1), it permits arbitrary levels of hierarchy within the logic block and it can give blocks different *modes* that represent significantly different functionality and interconnect of portions of the block. Modern commercial FPGAs have different modes in their memory blocks, for example, they can be configured as say 4Kx8, or 8Kx4, or 16Kx2 memories and so on. The new language permits the description of an FPGA with many different kinds of blocks, each of which can have the above features. The new language also allows the specification of timing for the atoms and their interconnect.

Secondly, we present a new area-driven packing framework and algorithm that takes as input a user design as well as an architectural description in the language mentioned above, and then determines area-efficient legal groupings of the atoms in the design into the logic blocks specified in the language. This problem is far more complex than the traditional LUT-packing problem [5] because of the non-simple interconnects, hierarchy and modes. Indeed, this kind of packing problem contains within it a combined placement and routing problem. Fortunately, the packing context permits some efficiencies which we describe in the paper.

We illustrate these new capabilities through two architectural experiments: one that explores different fracturable LUTs, now commonly used in industry, and one that explores different aspects of hard block memory architectures. In each of these experiments, we use a new set of large-scale Verilog circuits that contain both memory and multipliers.

This paper is organized as follows: the next section provides relevant background; Section 3 describes the new language with examples, and Section 4 gives the generic packing algorithm. The new features are integrated into an existing FPGA CAD system. Section 5 gives the illustrative architecture explorations. Section 6 concludes.

2. BACKGROUND AND TERMINOLOGY

The architecture of an FPGA consists of the set of blocks that perform internal computing, the input/output blocks that communicate with the extra-chip environment, and the programmable routing structure that connects them. As FPGAs have evolved, they have employed increasingly more complex logic blocks that consist of a larger number of small components, which we will call *primitives*, grouped together. One purpose of this grouping, often called *clusters*, is to leverage the locality typically found in circuits. This section describes the prior work on languages that describe such complex blocks and algorithms that pack a user circuit into complex blocks.

2.1 Complex Block Architecture Description Languages

In order to explore the large space of complex block architectures, a language that can precisely specify those complex blocks is needed. Over the years, several languages have been developed that target different trade-offs between expressiveness and conciseness for complex blocks.

Some languages gain conciseness by limiting the complex block architectures that they can describe to a restricted subset and then employ parameters to select between different instances of that subset. These languages include those used in VPR 4.30 [6] and VPR 5.0 [20] which target a simple block consisting of a cluster of fully-connected basic logic elements and Ho's language [13] which describes more sophisticated floating-point cores as blocks.

Other languages focus more on expressiveness. Cronquist's Emerald [10], Filho's CGADL [12], Ebeling's language for RaPiD [11], and the languages described in [26], use a netlist representation which, though very expressive, is cumbersome and verbose when expressing simple soft logic complex blocks. Paladino proposed a general complex block description language called CARCH in [29] which employs properties and rules to gain expressiveness, but was focussed on more microscopic attributes of common soft logic blocks.

2.2 Packing Algorithms

There is large body of prior work on the packing problem for FPGAs. Most of it focuses on the optimization of area, delay, and/or power for the basic (LUT-based) soft logic complex blocks. These algorithms include T-VPack [21], T-RPack [7], IRAC [30], HDPack [9], and others [17] [18]. Lemieux [16] and Wang [32] investigated packing to a basic soft logic complex block that contains a depopulated crossbar.

There has also been work on packing for complex blocks that are significantly different from the basic complex block. Ni proposed an algorithm that packs together netlist blocks for clusters with arbitrary interconnect and an arbitrary number of heterogeneous primitives [28]. The algorithm does not scale and it is intractable to use it to model all but the smallest soft logic clusters. Ahmed described packing DSP blocks to make use of regularity in placement and routing [4]. Paladino proposed a design rule check (DRC) based packer called DC that can pack to the soft logic of two different Altera FPGA families [29]. Limitations with the tool prevent it from exploring non-trivial complex blocks such as memories and fracturable LUTs.

3. A NEW COMPLEX BLOCK ARCHITECTURE DESCRIPTION LANGUAGE

A key goal of this work is to enable architecture exploration and CAD tool research for FPGAs with far more complex logic and interconnect than has been possible with prior public tools. In this section, we describe a new modeling language that permits the description of logic blocks with an arbitrary amount of hierarchy, that permits complex specification of the interconnection between logical elements, and that allows the specification of different modes of operation. To be as easy to use as possible, we seek to have the language be:

- Expressive: The language should be capable of describing a wide range of complex blocks.
- Simple: The language constructs should match closely with an FPGA architect's existing knowledge and intuition.
- Concise: The language should permit complex blocks to be described as concisely as possible.

In the following sections, we provide an introduction to the new language that shows how these goals are met. Due to space limitations, the full language itself, with detailed examples, is presented at
`http://www.eecg.utoronto.ca/vpr/arch_language.html`.
This language will be supported in the next release of VPR.

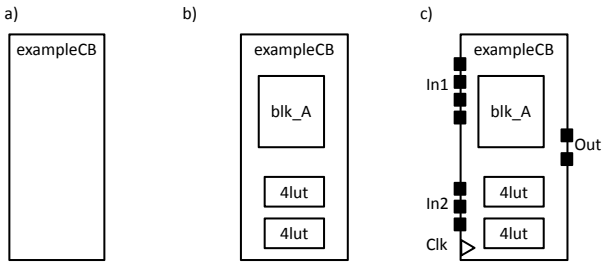

Figure 2: Example physical blocks

3.1 Overview

The new language uses XML syntax; readers unfamiliar with XML should review [31]. It also incorporates constructs that directly correspond to the hardware structures that most commonly occur in FPGA complex blocks – such as muxes and LUTs.

At the highest-level, the language contains two categories of construct: 1) physical blocks, and 2) interconnect. Physical blocks are used to represent the core logic, computational, and memory elements within the FPGA. This includes LUTs, flip-flops and memories. Interconnect constructs represent connectivitiy within and between physical blocks, including wiring, programmable switches, and multiplexers. We begin by describing the physical block construct.

3.1.1 Physical Blocks

The basic physical block type in the language is specified using the XML element *pb_type* which has a name attribute to identify it. To obtain the ability to describe arbitrary hierarchy, any *pb_type* start-end block can contain other *pb_type* specifications. For example, consider the empty complex block shown in Fig. 2 a). It is specified by the following code:

```
<pb_type name="exampleCB">
</pb_type>
```

A more complex block is shown in Fig. 2 b), which has three child blocks in it, two of the same type (labelled 4lut) and one different block (labelled blk_A). The language construct *num_pb* can be used to specify the number of instances of a child physical block that are contained in its parent physical block. The full specification of the example complex block in Fig. 2 b) is as follows:

```
<pb_type name="exampleCB">
  <pb_type name="blk_A" num_pb="1">
  </pb_type>
  <pb_type name="4lut" num_pb="2">
  </pb_type>
</pb_type>
```

Physical blocks must communicate with one another, and also with other blocks at the same level, as well as the external inter-block routing. A physical block will have a combination of input, output, and/or clock ports. A port comprises of one or more pins. The input, output, and clock ports are described using XML tags *input*, *output*, and *clock*, respectively. Each tag is declared as a child element of the *pb_type* on which the ports reside. Each port tag must be given an identifier with the *name* attribute. The number of pins associated with a port is specified with the *num_pins* attribute. For example, the block shown in Fig. 2 c) adds four ports to the complex block of part b), and its language specification is given below. Notice that the In1 port has four pins; the In2 port has three pins; the Out port has 2 pins; and, the Clk port has a single pin.

```
<pb_type name="exampleCB">
  <input name="In1" num_pins="4"/>
  <input name="In2" num_pins="3"/>
  <output name="Out" num_pins="2"/>
  <clock name="Clk" num_pins="1"/>
  <pb_type name="blk_A" num_pb="1">
  </pb_type>
  <pb_type name="4lut" num_pb="2">
  </pb_type>
</pb_type>
```

3.1.2 Modeling Primitives

Primitives are physical blocks at the bottom level of hierarchy – they do not contain other physical blocks. A primitive corresponds to the elements present in the technology-mapped user netlist, prior to the packing phase. The language attribute, *blif_model* must be included in the primitive *pb_type* element, and it specifies the type of input user netlist block that the primitive implements. The packer, described below, uses BLIF as the netlist format. The value of the *blif_model* attribute for a primitive pb_type is a string that should exactly match the string in BLIF used for the netlist block that can reside in the primitive.

The new language incorporates special handling for three of the most common types of primitives found in FPGAs: flip-flops, LUTs, and memory. We chose to do this to make it easier to deal with specific features of these primitives. The language *class* attribute is used to identify these primitives. Consider again the example in Fig. 2 b): we make the *4lut* a LUT primitive type by adding the *blif_model* and *class* attributes as follows:

```
<pb_type name="exampleCB">
  <input name="In1" num_pins="4"/>
  <input name="In2" num_pins="3"/>
  <output name="Out" num_pins="2"/>
  <clock name="Clk" num_pins="1"/>
  <pb_type name="blk_A" num_pb="1">
  </pb_type>
  <pb_type name="4lut" num_pb="2"
           blif_model=".names" class="lut">
  </pb_type>
</pb_type>
```

In processing the input user netlist, the BLIF construct *.names* is assumed to map into a LUT.

In addition to the *class* attribute, the ports on these primitives must be declared with a special attribute called *port_class* which provides necessary information about the pins on these special types of primitives, as described below:

1. *lut*: The LUT primitive has one port class for its inputs (called *lut_in*) and one for its output called *lut_out*. This is useful for example, so that downstream tools can take advantage of input pin *swapability*: signals on LUT inputs can be permuted and the LUT's truth table re-programmed accordingly. Note that more complex LUTs, such as fracturable LUTs, are described as clusters; basic LUTs within the more complex LUT are described using this LUT primitive.

2. *flipflop*: A flip-flop has three port classes: input (D), output (Q), and clock (*clock*), which have exactly one pin each. The library could be extended to support more ports for flip-flops (such as asynchronous clear).

3. *memory*: Single-port memories have three input port classes: *address*, *data_in*, and *write_en* and one output port class: *data_out*, which represent the related functionality of memories. Dual-port memories have six input port classes: *address1*, *data_in1*, *write_en1*, *address2*, *data_in2*, and *write_en2* and two output port classes: *data_out1* and *data_out2*. Both single and dual-port memories have one optional clock port class: *clock* (for synchronous memories). The library can be extended to support more ports for memories.

The following example describes a single-port memory primitive type to illustrate the usage of the *class* and *port_class* attributes:

```
<pb_type name="mem_1024x2"
         blif_model=".subckt single_port_ram"
         class="memory" num_pb="1">
  <input name="addr" num_pins="10" port_class="address"/>
  <input name="data" num_pins="2" port_class="data_in"/>
  <input name="we" num_pins="1" port_class="write_en"/>
  <output name="out" num_pins="2" port_class="data_out"/>
  <clock name="clk" num_pins="1" port_class="clock"/>
</pb_type>
```

It may occur to the reader that an alternative to introducing the *class* and *port_class* attributes would be to require that the architect give specific pre-defined names to pb_types and ports. We considered that approach, however, we deemed it overly restrictive. With the proposed class and port class scheme, the architect is free to name pb_types and ports any way he/she likes, which enhances readability and may ease integration with other tools that use different naming conventions.

3.2 Intra-Block Interconnect

The ports and pins on physical blocks are connected to one another using an *interconnect* element that is declared *within* a parent physical block type. There are three kinds of interconnect:

1. *complete*: This represents a complete crossbar switch from a set of inputs pins to a set of output pins. It is assumed that the particular input pin that is matched with a particular output pin is controlled by signals internal to the FPGA whose values are set during device configuration.

2. *direct*: This is a direct connection from one set of pins to another set of pins. This is used to model single metal wires or buses that have no programmability or switching.

3. *mux*: This is a multiplexed connection of single or multi-bit (bus) signals. As in the case of *complete*, it is assumed that signals internal to the FPGA (likely driven by configuration bits) control the select inputs of the multiplexer. That is, this construct represents a bus-based multiplexer whose input-to-output path is set during FPGA configuration.

The input and output pins of interconnect elements are specified by one *input* attribute and one *output* attribute declared within the interconnect element. The *complete* element has one set of pins for its input and one set of pins for its output. The *direct* element has one set of pins for its inputs and another set of pins for its outputs. The *mux* element has multiple sets of pins for its input and one set of

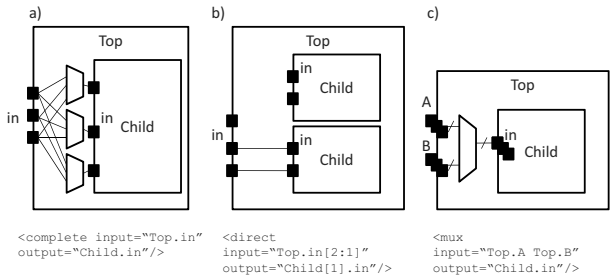

Figure 3: Examples interconnect types: a) complete b) direct c) mux

pins at its output. Each set of input pins is delimited by a space.

A set of pins to be connected is specified by first selecting physical blocks that are to be connected, and then specifying the desired pins on those blocks: In the case of there being multiple instances of a physical block, the following syntax is used:

```
<pb_type name>[<start index of physical block>:
  <end index of physical block>]
```

Physical blocks are indexed from 0 to *num_pb - 1*. If only one physical block is selected, then the colon and ending index may be eliminated. If there is only one physical block, then the entire *[* to *]* specification is not needed.

The pins on the block are specified in the following way:

```
<physical block port name>[<start index of pins>:
  <end index of pins>]
```

Pin indices start from 0 and end at *num_pins - 1*. There is one shortcut for pin selection: if the architect wishes to select all the pins of a port, then he can skip the section from *[* to *]*.

Fig. 3 gives examples of the three interconnect constructs. Underneath each figure is the code that produces the corresponding interconnect. The examples assume that interconnect connectivity is *from* pins on a physical block called *Top* to pins on one of Top's child physical blocks. For the *complete* interconnect case (in Fig. 3 a)), there is one physical block for each *pb_type* so only the *pb_type* is specified when selecting the blocks. All pins of the ports are used, so only the names of the ports are specified.

For the *direct* interconnect example (in Fig. 3 b)), only the last two of the three *Top.in* pins are used so the corresponding code specifies the range of pins using *[2:1]*. There are two physical blocks of type *Child* and only the one with index 1 is used, so the code includes a *[1]* in *Child[1]* to identify that block. This specification creates a one-to-one mapping between two input pins of *Top* and two input pins of *Child[1]*.

The *mux* interconnect example specifies a 3-bit 2-to-1 mux (in Fig. 3 c)). The input attribute to the mux has two 3-bit pin sets. The first pin set is *Top.A* and the second pin set is *Top.B*. The two pin sets are separated by a space. The output of the mux is one 3-bit pin set of *Child.in*.

For ease-of-use, the language provides a mechanism to concatenate sets of pins together. It follows a similar syntax to the concatenate construct in Verilog [8].

A "scope" question naturally arises with the use of the interconnect element: in an arbitrary multi-level hierarchy of physical blocks, which ports/pins can be used within an interconnect element that is declared within a physical block

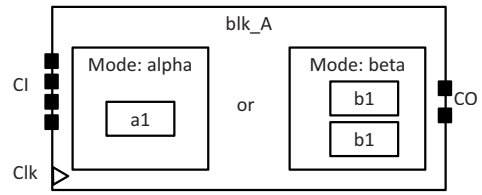

Figure 4: Example of a physical block with multiple modes of operation

at some specific level of the hierarchy? We take a straightforward approach to interconnect scope – the interconnect element can use pins of its parent physical block, or can use pins of any physical block declared in the same level of the hierarchy.

3.3 Modes

A physical block in an FPGA may have multiple modes of operation and such modes are normally mutually exclusive. For example, consider an FPGA memory block that can be configured with different aspect ratios such as 512x8 and 1024x4 [27]. Each of these different configurations needs to be represented by a unique mode of operation. To represent the mode concept, the language allows the definition of one or more *mode* elements within the *pb_type*. Multiple modes of operation are represented by multiple sibling *mode* elements declared within a parent *pb_type*. If a mode is declared, child physical blocks and interconnect can be declared *inside* the mode element, representing blocks (and connectivity) that is specific to the particular mode. In general, modes represent different ways of using a given piece of underlying FPGA hardware. A mode has one attribute *name* that serves as an identifier. Fig. 4 shows a physical block with multiple modes of operation. The first mode is called *alpha* and it contains one physical block *a1* and the second mode is called *beta* and contains two physical blocks of type *b1*. The corresponding code is:

```
<pb_type name="blk_A">
  <input name="CI" num_pins="4"/>
  <output name="CO" num_pins="2"/>
  <clock name="Clk" num_pins="1"/>
  <mode name="alpha">
    <pb_type name="a1" num_pb="1">
    </pb_type>
  </mode>
  <mode name="beta">
    <pb_type name="b1" num_pb="2">
    </pb_type>
  </mode>
</pb_type>
```

Different modes can each have their own unique interconnect by declaring one or more *interconnect* elements as children of a *mode* element.

Using these language constructs, we can model complex logic structures, including the one shown in Fig. 1. Due to space limitations, examples of how to model this and other logic structures are found in the website provided earlier.

4. PACKING ALGORITHM

In this section, we introduce our architecture-aware packing algorithm, *AAPack*. We begin with a top-level overview of the algorithm, and then elaborate on each step.

```
1: while (unpacked_netlist_blocks_exist())
2:    s = seed_netlist_block()
3:    B = new_complex_block(s)
4:    while (attempt_more_packs(B))
5:       c = candidate_netlist_block(B)
6:       attempt_pack(c,B)
7:    add B to output packed netlist
```

Figure 5: Generic iterative packing algorithm.

4.1 Overview

The input to the packer is a technology mapped netlist of unpacked netlist blocks, as well as a description of an FPGA architecture (specified in our language). The output is a netlist of *packed* complex blocks that implements the same functionality as the input netlist. Fig. 5 gives pseudocode for a generic iterative packing algorithm closely resembling those in published literature. The outer `while` loop at line `1` continues until all input netlist blocks are packed into complex blocks. At line `2`, a seed netlist block, s, is selected for a new complex block. Line `3` creates a new complex block, B, containing the seed block. The algorithm then proceeds to pack additional netlist blocks into B (inner loop on lines `4`-`6`). The loop on line `4` continues until no further packs into B should be attempted. Line `5` identifies a netlist block, c, that is a candidate for packing into B. Line `6` attempts to pack c into B, which presents unique challenges owing to the range of complex block architectures that can be described in our language. The process of finding additional netlist blocks to pack into B continues iteratively until either: 1) B is full, or 2) no such primitives are found. B is then added to the output packed netlist (line `7`) and control returns to the outer loop.

The algorithm in Fig. 5 represents a core packer engine that calls several functions for which a variety of implementations are possible. We elaborate on our initial implementation choices below.

4.2 Selecting Netlist Blocks and Complex Blocks

To choose the seed netlist block, s, for a new complex block (line `2` in Fig. 5), we borrow the approach of [5] and choose s to be the unpacked block with the largest number of nets attached.

Having initialized a new complex block with a seed netlist block, we use an affinity metric to select additional netlist blocks to pack into the complex block (line `5` of Fig. 5). Consider a netlist block p and a partially filled complex block B. The affinity between p and B is defined as:

$$Aff = \frac{(1-\alpha)\cdot nets(p,B) + \alpha \cdot connections(p,B)}{num\_pins(p)} \quad (1)$$

where $nets(p,B)$ is the number of shared nets between p and B, and $connections(p,B)$ is tied to the number of pins on p's attached nets that lie outside of B:

$$connections(p,B) = \frac{1}{ext(p,B) + packed(p) + 1} \quad (2)$$

where $ext(p,B)$ represents the sum total of pins on p's nets that reside on netlist blocks *not* packed into B, and $packed(p)$ represents the total number of pins on p's nets that attach to netlist blocks already packed into *other* complex blocks (aside from B). Connections between p and netlist blocks that are already packed are guaranteed to be inter-block connections in the packing solution, and thus, they are penalized in (2). Observe that $connections(p,B)$ is a pin-based count, whereas $nets(p,B)$ is a net-based count. The $num\_pins(p)$

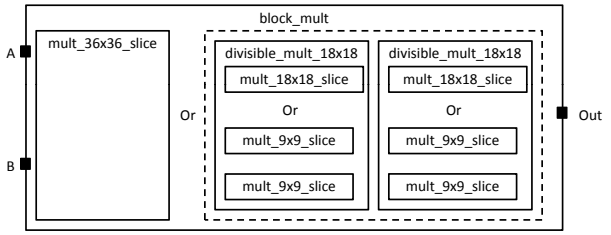

Figure 6: Reconfigurable multiplier example.

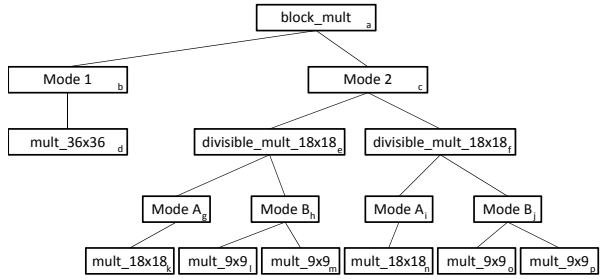

Figure 7: Tree representation of a complex block.

in the denominator of (1) is the number of used pins on p and it serves to normalize affinities across netlist blocks with different numbers of used pins. Parameter α in (1) is a scalar weight that we set to 0.9 for our experiments.

We use the affinity metric (1) to choose packing candidates in the `candidate_netlist_block` function in Fig. 5, preferring to pack together netlist blocks with high affinity for one another. Blocks that, based on their type, cannot be accommodated in the current complex block being packed are automatically filtered out from consideration by the `candidate_netlist_block` function. For example, we never consider packing a LUT into a multiplier-based complex block.

4.3 Legality Checking in Packing

Our packer must be able to handle arbitrary hierarchy and arbitrary interconnect – requirements that ultimately impact the `attempt_pack(c,B)` function in Fig. 5. The function performs two key tasks: 1) it finds a candidate primitive (location) for a netlist block c within a hierarchical complex block B; and, 2) for the chosen location, it ensures that the packing is *legal* from the perspective of routing c's nets through B's interconnect. These steps are analogous to placing and routing the netlist blocks within a complex block.

4.3.1 Location Assignment

For the purposes of packing, we represent a hierarchical complex block as an ordered tree. Nodes in the tree correspond to physical blocks or modes. Edges between nodes represent the parent/child relationships between physical blocks and/or modes. The root node of the tree corresponds to an entire complex block. Leaf nodes correspond to primitives within a complex block. Fig. 7 gives the tree representation for the multiplier-based complex block illustrated in Fig. 6. Observe that the tree is ordered: fine-grained physical blocks are on the right; coarse-grained physical blocks are on the left. The ordering allows us to traverse the tree according to non-decreasing physical block size. In general, it is desirable to pack a netlist block into the smallest primitive that can accomodate it (doing otherwise would likely result in poor utilization) – the tree ordering assists us in meeting this objective.

Given a candidate netlist block, c, and the tree representation of a complex block, we use a depth-first search to identify a location for c. Our search prioritizes exploring right children before left children. That is, we explore fine-grained physical blocks before coarse-grained blocks. As an example, consider a technology mapped netlist containing two multiplier blocks: a 16x16 multiplier and an 8x8 multiplier. We wish to pack the netlist into the complex block shown in Fig. 7, starting with the 16x16 multiplier block. Referring to the labels on each node, the search traverses downwards from the root, visiting nodes a, c, f and j. Since mode B at node j cannot accommodate the 16x16 block, the search backtracks to node f and continues until a feasible location is discovered, eventually packing the multiplier into node n. At this point, a routability check must be performed (described below). If the check is unsuccessful, the depth-first search continues for an alternative location. If the check is successful, we move onto the task of packing the next block, the 8x8 multiplier.

When a netlist block is successfully packed into a tree node, the depth-first search *pops up* to the node's parent, which corresponds to a sub-tree of the complex block hierarchy. We then look for netlist blocks that can pack into that sub-tree (using depth-first search). We try to fill up the sub-tree before proceeding to explore other parts of the hierarchy. Note that the `candidate_netlist_block` function filters blocks according to the current sub-tree being packed. For example, if we packed a LUT into a sub-tree comprising a LUT/flip-flop pair, our algorithm would attempt to find a flip-flop that can be packed together with the LUT. We attempt up to 30 packs on a sub-tree before the depth-first search pops up to explore other sub-trees.

4.3.2 Ensuring Routability

Simply finding a primitive for a netlist block within a complex block is not a sufficient condition for feasible packing. Our language allows the description of arbitrary interconnect within a complex block, and therefore, we must check that the netlist block's nets can be routed, both in the context of the interconnect specified for the complex block, and also in the context of other netlist blocks that are already packed into the same complex block. Each netlist block packed into a complex block may have connections to other netlist blocks packed into the same block and we must ensure there is sufficient intra-block interconnect for such connections. Likewise, a netlist block may have connections to netlist blocks packed in other complex blocks. Such connections will be routed through the general FPGA interconnect fabric. We must ensure that for such connections, there is a path to a top-level complex block pin (a "way out" of the complex block).

To assess routing feasibility, we first execute a basic check regarding whether packing the candidate netlist block into the current placement within the complex block causes the pin demand to exceed the available pins of any parent blocks. If this check fails, the candidate block is disqualified from packing into the complex block. Otherwise, we move onto a more rigorous routing assessment.

We model the complex block interconnect using a routing graph. A node in the routing graph represents a pin on a physical block in the complex block hierarchy. Directed edges between nodes correspond to paths through complex block interconnect. For the set of nodes corre-

sponding to the top-level complex block output pins, we create directed edges to the nodes corresponding to the top-level complex block input pins. In so doing, we model the ability for a primitive to connect to another primitive in the same complex block through the general FPGA interconnect fabric. We assume that the general FPGA interconnect is rich enough to allow any output pin on a complex block to connect to any other input pin on a complex block (as is generally the case for commercial FPGAs). This assumption eases the routing problem, as it means that external connections to (from) a primitive can be routed from (to) any top-level output (input) pin node of the complex block.

Having formulated the routing problem using a routing graph and a set of required pin-to-pin connections to route, we directly apply the PathFinder negotiated congestion routing algorithm [22] to determine if a feasible routing solution can be found. The maze router used within our PathFinder implementation is undirected (breadth-first), as packing does not incorporate a notion of geographical proximity

4.4 Handling Memories

Memories present a unique challenge for packing. The user's design may contain memories that are wider and/or deeper than the size of a physical memory block in the target FPGA. In such cases, multiple memory blocks in the FPGA are needed to implement the user's memory. The AAPack algorithm requires that memories in the input netlist be specified as one-bit-wide memories of depth not exceeding the depth of the largest physical memory in the FPGA[1]. For example, if the user's design contains a 256 x 8 memory, the packer's input will contain eight 256 x 1 memories that, ultimately, may be packed together in a single RAM primitive. In other words, for a memory instance in the user design, the total number of netlist blocks to represent that memory is equal to the word width of that memory instance. Memory primitives are thus handled differently than all other types of primitives in the sense that more than one memory netlist block in AAPack's input netlist may pack into a single memory primitive.

For memory blocks in the input netlist to be packed together into one physical memory primitive, two requirements must be met: 1) the memory blocks in the netlist must have the same address bus width, and 2) the signals on corresponding bits of the address bus and control signals must be identical.

4.5 Limitations of the Packing Algorithm

As is apparent from the discussion above, in our initial release, we have focused on area-driven packing. An implementation of timing-driven packing requires a detailed delay model for complex block interconnect and logic. Work is underway on this front and timing-driven packing will be included in a future tool release (the packer currently reads in timing and capactitance information but does not act on this information).

While the intent of AAPack is to provide good quality results for *any* complex block architecture, it is difficult to demonstrate this capability for the universe of architectures that can be modeled in our language. In this study, we have limited the types of complex blocks investigated to the following: 1) LUT-based complex blocks (including blocks with fracturable LUTs), 2) fracturable multipliers, and 3) memories with reconfigurable aspect ratios. Such types of complex blocks are pervasive in commercial FPGAs, yet they are unsupported by any public-domain packer.

We have also placed an architectural constraint that different complex block types cannot legally accomodate the same netlist block. For example, we do not investigate architectures where flip-flops can be packed into either a LUT-based complex block or a multiplier-based complex block. This constraint makes the choice of complex block type based on a netlist block straightforward – there can be only one complex block type that can accomodate a particular netlist block. We acknowledge that commercial FPGA packing does not have this limitation and we plan to remove this limitation in the future.

[1]The upstream RTL synthesis tool, ODIN II [14], splits memories to ensure that this requirement is met.

5. EXPERIMENTS

In this section, we describe the methodology and experiments to illustrate the new FPGA architecture language's ability to model and enable exploration of more complex blocks than in the past. This is done first by modelling and exploring soft logic blocks containing fracturable LUTs, and then block RAMs of different sizes with different configurable aspect ratios. We also evaluate the quality of the new generic packing algorithm, against a lower bound computation. We also compared our packing algorithm against a previous algorithm on a legacy architecture [19]; however, the results are omitted here due to a lack of space.

The complete CAD flow consists entirely of publicly-accessible source tools. We use ODIN II [14] for front-end HDL parsing, elaboration and partial synthesis. The ABC framework [25] is used for technology-independent optimization (using the `resyn2` script) and technology mapping. Circuits are mapped to minimize area using WireMap [15], implemented within ABC's priority cuts-based mapper [24]. Technology mapping is executed with *choices* [23] – an approach for reducing structural bias whereby mapping is done concurrently on multiple functionally equivalent circuit representations and the best mapping result is selected. Large circuit blocks (e.g. block RAMs and multipliers) are passed through ABC as black box modules. Note that the description of the complex blocks themselves are not used by ABC during logic synthesis but rather used afterwards during packing. Packing is performed by the algorithm described in Section 4. We use a modified version of VPR 5.0 [20] for non-timing-driven placement and routing. The packer is integrated into the VPR source code. The routing architecture is held constant for all experiments, and consists of single-driver length-4 wire segments, $F_s = 3$, and $F_c(in) = 0.15$ and $F_c(out)$=0.125, as per the usual nomenclature. Around the chip perhiphery, we assume there to be 7 I/O tiles per complex block column/row.

We employ a new set of benchmark circuits, as described in Table 1. This new suite of circuits contain block RAMs and multipliers of various sizes and were collected from a variety of sources. They include soft processors, video image processors, and range in size from 256 to 24,587 6-input LUTs. The columns in this table describe the name of the circuit, followed by the physical resources demanded by each circuit after technology mapping, including: the number of flip-flops, the total number of 6-input (or less) LUTs, the total number of memory bits, the number of logical memories, the maximum depth and width across those memories, and the number of multipliers.

Table 1: New Benchmarks and statistics.

Circuit	FFs	LUTs	Bits	#Mem	Max Depth	Max Width	#Mult
boundtop	1620	2779	32768	1	1024	32	0
ch_intrinsics	233	402	256	1	32	8	0
mkDelayWorker	2440	5046	532916	9	1024	313	0
mkPktMerge	36	256	7344	3	16	153	0
mkSMAdapter	952	1706	4456	3	64	61	0
or1200	611	2369	2048	2	32	32	1
raygentop	1185	1938	5376	1	256	21	18
reed_solomon	1591	3096	30720	15	256	8	0
stereovision0	12628	12318	33554432	1	524288	64	0
stereovision1	9558	10563	33554432	1	524288	64	152
stereovision2	13670	24587	33554432	1	524288	64	564

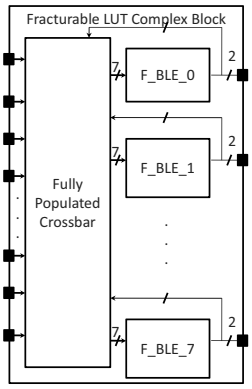

Figure 8: Global Structure of Fracturable Complex Block

5.1 Fracturable LUT Experiments

We demonstrate the utility of the new language and packer by using them to model complex blocks that contain *fracturable* LUTs. A fracturable LUT can be broken into two smaller LUTs that share input pins. Modern commercial FPGAs incorporate fracturable LUTs for the purpose of improving logic density, because many circuits synthesized by tools contain small LUTs that can be paired together and implemented in a single fracturable LUT.

Fig. 8 shows the global structure of a complex block based on fracturable LUTs, which looks similar to a classical cluster, except that each BLE has two outputs. A fracturable BLE contains a fracturable (dual-output) LUT and two bypassable registers – one register for each LUT output. Fig. 9 shows a fracturable BLE with 7 inputs and bypassable registers. A fracturable LUT has two modes of operation: 1) as a single K-input LUT, or 2) as two LUTs that together use at most FI inputs. In the dual-LUT mode, parameter FI determines the amount of pin sharing that is required between the pair of LUTs that are implemented in the fracturable LUT. Fig. 10 shows an example of a fracturable LUT. This fracturable LUT can operate as either one 6-LUT ($K = 6$) or two 5-LUTs that share 3 inputs ($FI = 7$).

A key architectural question for fracturable LUT architectures concerns the selection of the value for FI. Larger values for FI will permit more packing flexibility at the cost of more pins, whereas lower values of FI will reduce packing flexibility. We explore this question for a base architecture with $K = 6$ (the LUTs have 6-inputs when used in single-output mode) and $N = 8$ (there are 8 fracturable BLEs per complex block). We vary FI from 5 to 10, covering all possible pin sharing amounts from all to none. The meaning of $K = 6$ and $FI = 5$ requires elaboration: when the LUT

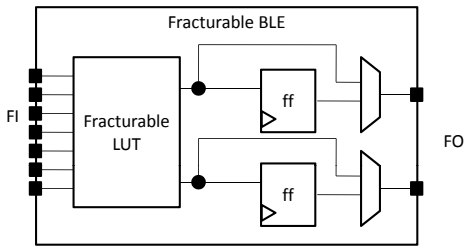

Figure 9: A fracturable BLE with 7 inputs, 2 outputs, and optional output registers

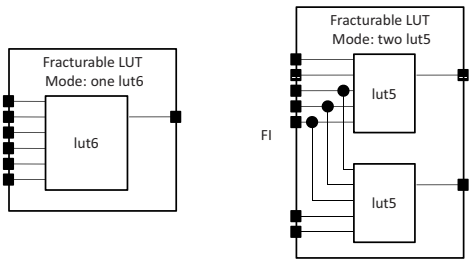

Figure 10: Two modes of a fracturable 6-LUT with 7 inputs.

is used in dual-output mode, the two LUTs are allowed to use no more than 5 distinct input signals (as is the case for Virtex 6 [3]). We also note that the number of inputs to the full complex block itself, I, is set equal to $FI \times N$, which implies that no pin sharing requirements are imposed *between* BLEs within the complex block. This architectural choice creates some architectural side-effects as described in the results below.

We evaluate the effectiveness of various fracturable LUT architectures by comparing the number of complex blocks in packing solutions, and with a lower bound on the optimal number of complex blocks needed. The lower bound is computed as follows:

```
# CB lower bound = ceiling((
   (# 5-LUTs or smaller + # unabsorbable FFs) / 16)
   + (# 6-LUTs / 8))
```

The "# 5-LUTs or smaller" count is the number of LUTs in the input to the packer that use 5 or fewer inputs. The "# unabsorbable FFs" is the number of flip-flops that structurally cannot be packed with a LUT (such as flip-flops fed by memories). The # 6-LUTs is the number of LUTs that use *exactly* 6 inputs. The bound was developed through a counting argument: There are 8 fracturable BLEs in a complex block. Each fracturable BLE can implement one 6-LUT, so the number of 6-LUTs in a design increases the complex block count by 1/8. Each fracturable BLE can alternatively implement (*at most*) two 5-LUTs, so each LUT in a benchmark circuit that uses 5 or fewer inputs increases the complex block count by 1/16. Flip-flops that cannot structurally be packed into a LUT prohibit a 5-LUT from being used so they increase the complex block count by 1/16. With this lower bound we can define the *logic efficiency* for fracturable LUT-based complex blocks packing as follows:

```
efficiency = # CB Achieved / # CB lower bound
```

Fig. 11 illustrates the result of packing the circuits in Table 1 into different fracturable LUT architectures with the parameters cluster size $N = 8$, and LUT size $K = 6$, with

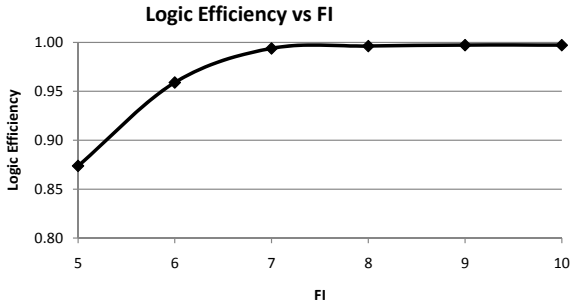

Figure 11: Logic efficiency vs. Number of Inputs to Fracturable LUT (FI).

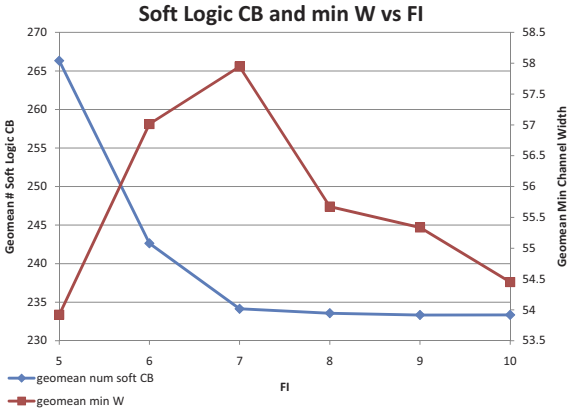

Figure 12: Number of Soft Logic CB and Min W vs. Number of Inputs to Fracturable LUT (FI).

FI varied from 5 to 10. The Y-axis in the figure gives the logic efficiency as defined above. Each point is the geometric average across the 11 new benchmarks. We can observe several things from this figure: First, for some values of FI, the packer achieves 100% efficiency against the lower bound, giving us some confidence that it is working well. Second, there is a significant leap in efficiency when the value of FI moves from 6 to 7, suggesting that there are many situations in which there are smaller LUTs to pack in with larger LUTs. It is clear, from this data, that a value of 7 or possibly 6 for the number of inputs (FI) is sufficient for these circuits and architectures.

Fig. 12 illustrates the impact of varying FI on the number of soft logic complex blocks and the minimum achievable channel width after placement and routing. As expected, the number of logic blocks declines with increasing FI, as the block gains flexibility. The effect of FI on channel width is more complex, in part due to an architectural artifact described above: on the left hand side of the figure, channel width increases with FI as there are more pins being routed into the logic blocks. However, after FI = 7, the number of pins saturates at the maximum (as shown in Fig. 11). A this point, the fact that we continue to provide extra routing pins on the outer complex block only serves to make the routing problem easier, increasing flexibility and therefore lowering channel width.

5.2 Memory Architectures

The purpose of the second experiment we ran is to illustrate the new language and packer's ability to describe and explore different physical block memory architectures. The complete target FPGA architecture contains traditional

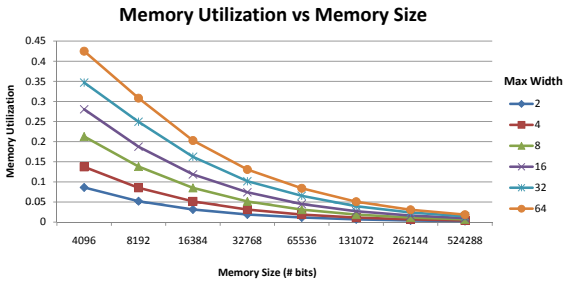

Figure 13: Memory utilization of architectures with varying physical memory sizes.

$N = 8/K = 6$ non-fracturable LUT soft logic blocks, fracturable 36x36 hard multipliers, and configurable hard memory blocks. The memories have a configurable aspect ratio, in which the width and depth can be traded-off, as is now common in commercial FPGAs. These are represented as different modes of the memory, as discussed above. The memories can also be configured to be either single-port or dual-port mode, with the maximum number of total data pins across each mode kept constant. This means that when the memory operates in dual-port mode, it can be at most half as wide as the widest single-port memory.

We vary two parameters of the physical memories: first, the number of bits contained in each hard memory. This is also the maximum depth of the memory when it is configured to have single-bit data width. Clearly fewer blocks would be needed as the size of the memory grows larger, but more of the bits will be wasted when those larger blocks are used to implement smaller memories. At the same time, smaller physical memories in the FPGA may require more soft logic multiplexers to glue together smaller memories into larger logical memories. The second parameter is the maximum data width of the configurable memory. For example, a 1024-bit memory with a maximum width of 8 can implement all powers of 2 width up to the maximum: a 1024x1, 512x2, 256x4, or 128x8 memory. If this number is too small, the memory architecture won't be flexible enough to efficiently use the memory bits. We again use the 11 benchmark circuits described in Table 1 in the flow described above.

Fig. 13 gives a plot of memory utilization (defined as the number of used memory bits divided by the total number bits in used memory blocks after packing) versus the size of the physical memory block in bits, for six values of maximum width, ranging from 2 to 64. This figure shows the expected trends, with utilization increasing as the physical memory size decreases. Also, as the maximum width increases, utilization gets better.

Fig. 14 gives the geometric mean of the number of soft logic complex blocks used (across all 11 circuits) as the physical memory size is varied. Clearly, for the smaller physical memories, the amount of soft logic needed to implement the multiplexers begins to grow significantly for memories less than 4K bits, at least for the logical memories demanded by our benchmarks. Although we show the count for only the case of max width = 64, these results are the same for all values of max width.

Taken together, Fig. 13 and Fig. 14 suggests the need for actually having at least two physical memory sizes - a small one to achieve good utilization on small memories and a larger one to prevent the inefficiencies of gluing together many small blocks.

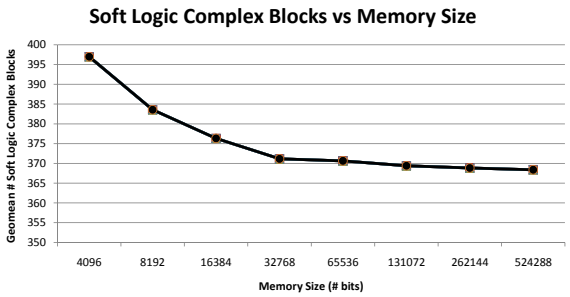

Figure 14: Soft logic complex block count with varying physical memory sizes.

6. CONCLUSIONS

We have presented a new FPGA logic block architecture description language that permits the modelling of far more complex soft logic blocks and hard logic blocks than was previously possible. The key features of the language are its ability to describe hierarchy, modes and arbitrary interconnect between atomic elements in the block. We have also presented a packing algorithm that begins to address the complexities of the FPGAs that use the new language, and shown it can be applied to explore block architectures that weren't previously explorable with public software - fracturable LUT-based architectures, and memory architectures. These are but a few of the blocks that can be studied with the new capabilities. We have demonstrated these capabilities with a new infrastructure capable of synthesizing circuits from Verilog, and with circuits that use memory and multipliers that are significantly larger than the previous standard benchmarks.

There is much more research and development remaining to flesh out the new capabilities: the packer must become timing-driven, and be enhanced to deal with the heterogenous case when one logical atom can be packed into two or more different complex physical blocks. We also need to explore more widely varying architectures and enhance the speed of the packer and quality of the packer for these architectures. Ultimately, we will employ this infrastructure to explore far more widely varying FPGA architectures from the area, speed and power perspectives.

7. REFERENCES

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

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

[3] Xilinx Virtex-6 FPGA Configurable Logic Block User Guide. `http://www.xilinx.com/support/documentation/user_guides/ug364.pdf`, September 2009.

[4] T. Ahmed, P. Kundarewich, J. Anderson, B. Taylor, and R. Aggarwal. Architecture-Specific Packing for Virtex-5 FPGAs. In *ACM Int'l Symp. on FPGAs*, pages 5–13, 2008.

[5] V. Betz and J. Rose. Cluster-Based Logic Blocks for FPGAs: Area-Efficiency vs. Input Sharing and Size. *IEEE Custom Integrated Circuits Conf.*, pages 551–554, 1997.

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

[7] E. Bozorgzadeh, S. Memik, X. Yang, and M. Sarrafzadeh. Routability-driven Packing: Metrics and Algorithms for Cluster-Based FPGAs. *Journal of Circuits Systems and Computers*, 13:77–100, 2004.

[8] S. Brown and Z. Vranesic. *Fundamentals of Digital Logic with Verilog Design*. Tata McGraw-Hill, 2007.

[9] 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.

[10] D. Cronquist and L. McMurchie. Emerald: An Architecture-Driven Tool Compiler for FPGAs. In *ACM Int'l Symp. on FPGAs*, pages 144–150, 1996.

[11] C. Ebeling, D. Cronquist, and P. Franklin. RaPiD Reconfigurable Pipelined Datapath. *Field-Programmable Logic Smart Applications, New Paradigms and Compilers*, pages 126–135, 1996.

[12] J. Filho, S. Masekowsky, T. Schweizer, and W. Rosenstiel. CGADL: An Architecture Description Language for Coarse-Grained Reconfigurable Arrays. *IEEE Trans. on VLSI*, 17(9):1247–1259, 2009.

[13] C. Ho, C. Yu, P. Leong, W. Luk, and S. Wilton. Floating-point FPGA: Architecture and Modeling. *IEEE Trans. on VLSI*, 17(12):1709–1718, 2009.

[14] P. Jamieson, K. Kent, F. Gharibian, and L. Shannon. Odin II-An Open-Source Verilog HDL Synthesis Tool for CAD Research. In *IEEE Annual Int'l Symp. on Field-Programmable Custom Computing Machines*, pages 149–156. IEEE, 2010.

[15] S. Jang, B. Chan, K. Chung, and A. Mishchenko. WireMap: FPGA Technology Mapping for Improved Routability. In *ACM Int'l Symp. on FPGAs*, pages 47–55. ACM, 2008.

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

[17] 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.

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

[19] J. Luu. A Hierarchical Description Language and Packing Algorithm for Heterogeneous FPGAs. Master's thesis, University of Toronto, Toronto, Ontario, Canada, 2010.

[20] J. Luu, I. Kuon, P. Jamieson, T. Campbell, A. Ye, W. M. Fang, and J. Rose. VPR 5.0: FPGA CAD and Architecture Exploration Tools with Single-Driver Routing, Heterogeneity and Process Scaling. In *ACM Int'l Symp. on FPGAs*, pages 133–142, 2009.

[21] 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.

[22] 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.

[23] A. Mishchenko, S. Chatterjee, and R. Brayton. Improvements to Technology Mapping for LUT-Based FPGAs. In *ACM Int'l Symp. on FPGAs*, pages 41–49, 2006.

[24] A. Mishchenko, S. Cho, S. Chatterjee, and R. Brayton. Combinational and sequential mapping with priority cuts. In *IEEE/ACM Int'l Conf. on CAD*, pages 354–361, 2007.

[25] A. Mishchenko et al. ABC: A System for Sequential Synthesis and Verification. `http://www.eecs.berkeley.edu/alanmi/abc`, 2009.

[26] P. Mishra and N. Dutt. Architecture Description Languages for Programmable Embedded Systems. *IEEE Proc. on Computers and Digital Techniques*, 152(3):285–297, 2005.

[27] T. Ngai, J. Rose, and S. Wilton. An SRAM-programmable field-configurable memory. *IEEE Custom Integrated Circuits Conf.*, pages 499–502, 1995.

[28] 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.

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

[30] 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.

[31] W3C. Extensible Markup Language (XML). `http://www.w3.org/XML/`, 2003.

[32] 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.

[33] S. Yang. Logic Synthesis and Optimization Benchmarks User Guide Version 3.0. *MCNC, Jan*, 1991.