2011 21st International Conference on Field Programmable Logic and Applications

RapidSmith: Do-It-Yourself CAD Tools for Xilinx FPGAs

Christopher Lavin, Marc Padilla, Jaren Lamprecht, Philip Lundrigan,
Brent Nelson and Brad Hutchings
NSF Center for High-Performance Reconfigurable Computing (CHREC)
Dept. of Electrical and Computer Engineering
Brigham Young University
Provo, UT, 84602, USA
Email: {chrislavin, brent_nelson, brad_hutchings}@byu.edu

***Abstract*—Creating CAD tools for commercial FPGAs is a difficult task. Closed proprietary device databases and unsupported interfaces are largely to blame for the lack of CAD research found on commercial architectures versus hypothetical architectures.**

This paper formally introduces RapidSmith, a new set of tools and APIs that enable CAD tool creation for Xilinx FPGAs. Based on the Xilinx Design Language (XDL), RapidSmith provides a compact, yet, fast device database with hundreds of APIs that enable the creation of placers, routers and several other tools for Xilinx devices. RapidSmith alleviates several of the difficulties of using XDL and this work demonstrates the kinds of research facilitated by removing such challenges.

I. Introduction

Broadly viewed, most FPGA research falls into one of two categories. One category tends to deal with commercial devices and often includes such topics as application-mapping experiments, application support, e.g., OS extensions, IP libraries, and limited CAD tool efforts, etc. The other category focuses on hypothetical architectures and includes architecture exploration and CAD tool efforts. Much of the work on hypothetical architecture and CAD tool research is facilitated by VPR [1]. However, studying and developing CAD tools, e.g., placers, routers, etc., for commercial devices is challenging or impossible. Device databases are often proprietary and the provided interfaces are either not well suited for CAD tool development, or they don't provide sufficient information to enable someone to write a router, for example.

The challenges of custom CAD tool creation for commercial FPGAs are quite unfortunate. Carrying out experiments on real FPGAs brings tremendous credibility to new concepts and ideas as they are proof of concept in their own right. Experiments targeting commercial FPGAs enable new abilities for researchers such as discovering and mitigating new FPGA security vulnerabilities, insights into bitstream density or its sensitivity to single event upsets in high radiation environments. If more researchers had the capability to create custom CAD tools for commercial FPGAs, it is likely that higher quality solutions would be found to several of the growing challenges faced by FPGA vendors.

This work was supported by the I/UCRC Program of the National Science Foundation under Grant No. 0801876.

We have encountered the challenges of creating custom CAD tools for commercial FPGAs first hand in previous work [2]. The goal of our previous work was to demonstrate the potential of rapid FPGA compilation techniques that could enable speedups of an order of magnitude or more for rapid prototyping purposes. At the onset of our previous work, we found commercial FPGA vendor tools to be insufficient to carry out our new ideas and techniques. Due to the lack of flexibility in vendor tools, we realized the necessity of creating custom CAD tools to accomplish our goal. At the time, no open tool or framework existed to create custom CAD tools for commercial FPGAs. With the lack of such a potentially useful tool, we undertook the initiative to create one ourselves in order to accomplish our goal.

The purpose of this work is to formally introduce RapidSmith, a new set of tools and APIs which enable FPGA CAD researchers to create and test new algorithms on current Xilinx FPGAs with unprecedented detail. RapidSmith is written in Java and has been released as an open source project. It is based on the Xilinx Design Language (XDL) and heavily leverages the Xilinx tool `xdl`. and the `xdl` tool enable: Unfortunately, XDL is not well understood by many and its full potential is often not realized. RapidSmith is able to overcome several of the challenges of using XDL which have likely hampered its use in the past. Although RapidSmith has been previewed in the past [3], this paper demonstrates its efficient device database structure and how RapidSmith makes it easy to create and customize tools such as placers and routers for Xilinx devices, further enabling and accelerating several branches of FPGA CAD research.

II. XDL: The Xilinx Design Language

RapidSmith is based on the interface provided by Xilinx called the Xilinx Design Language (XDL). The Xilinx tool `xdl` provides three important capabilities (options) to enable open FPGA CAD tools for Xilinx FPGAs:

1) `-report`: Generates complete FPGA device reports detailing placement sites and an exhaustive routing graph (without timing data)

978-0-7695-4529-5/11 $26.00 © 2011 IEEE
DOI 10.1109/FPL.2011.69

2) `-ncd2xdl`: Performs NCD to XDL conversion (allowing conventional Xilinx designs to be converted to the open XDL format)
3) `-xdl2ncd`: Performs XDL to NCD conversion (allowing XDL-manipulated designs to be re-injected into several locations within the Xilinx design flow)

With these options, the `xdl` tool provides detailed device information and can import and export designs to and from the Xilinx flow. This section serves as a short reference for XDL as it is no longer fully documented by Xilinx and a correct understanding of XDL is necessary in order to fully understand the value and capabilities of RapidSmith.

A. Detailed FPGA Descriptions in XDLRC Reports

The first option supplied by `xdl` is the `-report` option. This option takes a Xilinx part name as input and outputs an XDLRC report file which details all of the tiles, primitive sites and routing resources (without timing data) that are available in the given part[1].

1) Tiles: XDLRC report files abstract the FPGA fabric into a two dimensional array of sections called "tiles" which are conceptually laid out edge to edge in a checker board fashion. Each tile is specified with a name and a type. Tiles are mainly used as a reference to specify locations of certain resources. Tiles contain declarations of FPGA resources such as primitive sites, wires, and PIPs.

2) Primitive Sites: A primitive site is a location where an instance of an FPGA primitive can reside. Each primitive site has a type that is compatible with one or more primitive instance types. For example, a SLICEL is a type of primitive available in Virtex 4 parts. Half of all slice primitive sites in Virtex 4 parts are type SLICEL and the other half are type SLICEM. A SLICEL primitive can be placed on either a SLICEL or SLICEM site, however, the converse is not true. A list of primitive definitions is included at the end of every XDLRC report file which define all of the inputs, outputs and configurable internals of a primitive.

3) Wires: Wires declared in XDLRC reports refer to wires which traverse tile boundaries. A wire is simply unprogrammable metal that exists on the FPGA fabric as part of the routing graph. A wire always begins in a tile and can span one or more tiles. One of the peculiar attributes of XDLRC is that even though a wire represents a continuous piece of metal in the FPGA fabric, a separate name is given to each segment of the wire for each tile it occupies. Thus, a wire which spans 3 tiles will have three separate names attached to it depending on which tile is being referenced.

4) PIPs: Programmable interconnect points, or PIPs, are the configurable part of the FPGA routing graph. PIPs are completely contained within a tile (they do not straddle a tile) and define a potential connection that can exist between two wire segments. In most cases, thousands of PIPs are defined in the switch matrix tiles of an FPGA, most of which allow a certain wire segment to connect to one of many different wire segments.

[1]It should be noted that Altera provides an interface called QUIP [4] that enables replacement of parts of the CAD flow with custom tools. However, QUIP does not provide the detail necessary to construct a detailed router as XDL does.

B. Designs in XDL

The second and third options provided by `xdl` are the XDL/NCD conversion functions which enable proprietary Xilinx NCD files (native netlists for Xilinx FPGAs) to be converted to and from the open XDL format. A design in XDL is equivalent in most respects to Xilinx NCD files except that it is an ASCII file format that is human readable and it also is directly incompatible with other Xilinx tools in the flow.

XDL design files contain four types of declarations: design, module, instance and net. The first statement found in XDL designs is the design statement and occurs only once in each design file. It specifies the design name and part that the design is targeting. The module declaration is discussed in detail in Section IV-A2. The majority of the contents of an XDL design exist in the instances and nets.

1) Instances and Placement: All logic elements of a design are contained within the instances declared in the XDL file. An instance is an instantiation of a particular primitive type (SLICEL, SLICEM, etc.) that has a unique name, a type, an optional primitive site assigned to it and a list of attributes and values which configure the instance. Since assignment to a primitive site is optional, XDL files can represent designs that are unplaced, partially placed or fully placed. Placement of a design takes place when instances are assigned specific primitive site locations in the final implementation.

2) Nets and Routing: In order to specify connections between instances, the net declaration is used. A net has a unique name, a type, a list of input/output pins and optionally, a list of PIPs. Each net has one outpin (source) and one or more inpins (sinks). Pins are declared by using the unique instance name and pin on the instance. PIPs define how a net is routed and the presence of PIPs in a net indicate that the net is at least partially routed if not fully routed. Therefore, the task of a router is simply to assign a list of PIPs to a net. Nets cannot be routed if a design is not placed (PIPs require placed instances to make connections), however, XDL can represent designs that are un-routed, partially-routed or fully routed.

III. RapidSmith: Leveraging XDL for Rapid Creation of FPGA CAD Tools

There are several challenges that exist to successfully use XDL to build new CAD tools and leverage it as a design exchange method. Although XDLRC report files are extensive and provide a complete view of an FPGA architecture, they are extremely verbose text files that can exceed 70 gigabytes in size. This makes directly using XDLRC report files infeasible for creating design tools. Another challenge exists due to lacking information in the XDLRC report files that is necessary for efficient placement and routing. An open source XDL parser and data structure is not provided with the `xdl` tool making XDL manipulation often delegated to customized Perl scripts for every specific task.

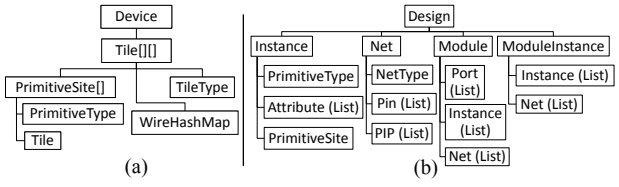


Fig. 1: RapidSmith abstractions for (a) devices and (b) designs.

RapidSmith provides two core packages to deal with the challenges of using XDL. First, RapidSmith provides a compact XDLRC-equivalent database file format and corresponding data structure (pictured in Figure 1a) that is fast loading and suitable for all kinds of applications. RapidSmith also includes the lacking information that can aid in making more effective placement and routing tools.

Second, RapidSmith provides a complete XDL parser and corresponding data structure (pictured in Figure 1b) to easily facilitate manipulating XDL designs and enable researchers to build highly customized CAD tools for Xilinx FPGAs.

A. Xilinx FPGA Database Files in RapidSmith

A very important attribute of an FPGA CAD database is its size and consequently, its load time from disk. RapidSmith employs three major strategies in order to control its device database size and load time; (1) aggressive wire and object reuse, (2) careful pruning of unnecessary wires, and (3) customized serialization with compression.

1) Wire and Object Reuse: Wires are the major reason for the gigantic size of the XDLRC report files. FPGAs are generally very regular replicated structures and the same wires can appear identically all over the chip. Efficiently accommodating the infrequent irregularities that occur in the routing structure is key to creating an efficient and much smaller device database. Therefore, rather than define a wire object for every wire in every tile, RapidSmith employs a unique set of wires that are specified by a name and a set of tile offsets that remove reference to any particular tile in the chip, making them reusable. This reusability factor is able to dramatically reduce the size required to represent wire connections on the device.

2) Wire Graph Pruning for Efficiency: The XDLRC report contains wires that are present simply for the sake of completeness, but, do not contribute more useful information to the actual routing structure. For example, in a HEX wire (a wire which spans 7 switch matrices), connections to the wire can only be made in 3 of 7 tiles which it spans. The 4 wire segments in tiles that do not connect to other wires are implied by the overall structure and thus, removed by RapidSmith. By removing these wires, significant memory savings are obtained as well as faster routing times as the removed wires are no longer examined when expanding connection searches.

3) Serialization and Compression: RapidSmith uses custom serialization to store only the essential data needed to reconstruct the device databases. The default Java serialization routines were considered, but were later abandoned as they turned out to be inefficient and caused files to load more slowly. RapidSmith uses the Hessian 2.0 serialization compression protocol [5] to further reduce the size of the device file. The Hessian protocol provides low-level serialization APIs and also compresses/decompresses the data found in the device files with minimal impact on load time.

TABLE I: RapidSmith Device Files Performance[1]

Xilinx Part Name	XDLRC Report Size	Compressed File Size	Java Heap Usage	Load Time From Disk
V4 SX55	3.5GB	539KB	34MB	0.299s
V4 FX140	8.0GB	1546KB	70MB	0.616s
V4 LX200	10.0GB	1010KB	61MB	0.602s
V5 FX200T	9.4GB	1227KB	60MB	0.585s
V5 TX240T	10.0GB	1111KB	56MB	0.620s
V5 SX240T	11.9GB	1135KB	61MB	0.630s
V5 LX330	12.5GB	1069KB	69MB	0.622s
V6 CX240T	8.5GB	937KB	35MB	0.460s
V6 SX475T	17.7GB	1506KB	61MB	0.814s
V6 LX760	22.8GB	1758KB	77MB	1.068s
V7 855T	32.0GB	2634KB	115MB	1.408s
V7 1500T	53.0GB	4985KB	263MB	2.653s
V7 2000T	73.6GB	5956KB	301MB	3.339s

[1]Measurements were recorded on a Windows 7 64-bit workstation with a Core i7-860 processor, 8GB of DDR3 RAM and 1TB 7200RPM SATA hard disk. The 32-bit Oracle JVM ver. 1.6.0_22 was used for Java bytecode execution.

4) Overall Performance: Using all of the techniques mentioned, RapidSmith database files are able to achieve a file size compression of over 10,000× when compared to the original XDLRC report files and all but the largest files can be loaded in less than a few seconds. A summary of large parts and their corresponding device file statistics are shown in Table I.

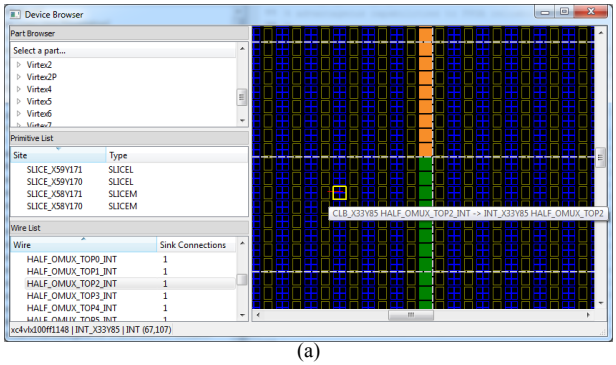
B. Augmented XDLRC Information in RapidSmith

The XDLRC report neglects to inform the user about primitive sites that can support more than one primitive type. The lack of this information provides two inefficiencies or challenges: (1) placements produced without this information may not be complete or may result in inefficient placements and (2) certain primitives do not have native sites of their own and therefore must reside on sites of a different type. When a primitive resides on a site of a different type, there can be pin name mis-matchings that can cause problems for a router.

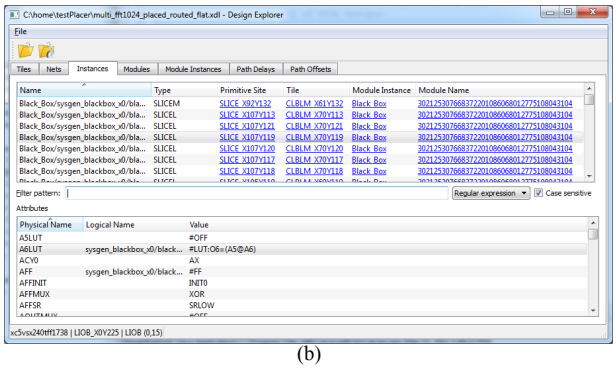
Using RapidSmith, we have built special programs to automatically compile a complete listing of primitive site compatibilities and pin name mappings. This information is included with the RapidSmith distribution and integrated into the APIs to provide a seamless solution.

C. XDL Design Representation in RapidSmith

As already shown, Figure 1 illustrates how RapidSmith data structures represent the data contained in XDL. Each class in RapidSmith will often represent a one-to-one relationship with XDL/XDLRC constructs that make using RapidSmith intuitive to use with XDL designs and XDLRC reports. A fully optimized XDL parser and export method are included with RapidSmith to populate its data structures and produce Xilinx-compatible XDL for re-injection into the Xilinx flow. RapidSmith also includes graphical tools to browse and explore devices and designs as shown in Figure 2. The Device



(a)



(b)

Fig. 2: Graphical tools provided with RapidSmith to browse (a) devices or (b) designs.

Browser, shown in Figure 2a, illustrates how the user can explore the resources and wires available in a device in RapidSmith and demonstrates the framework which can be used by researchers to create custom graphical CAD tools that interact with the FPGA fabric. The Device Browser leverages the classes outlined in Figure 1a. The Design Explorer application, shown in Figure 2b, allows the user to load XDL designs and, optionally, timing reports to rapidly search through the different constructs and leverages the classes found in Figure 1b.

The APIs available in the design package of RapidSmith are particularly rich with over 700 documented methods for design manipulation. These methods include functionality such as placement/un-placement of instances, creating new instances from scratch, determining the type of nets (wire, VCC, GND, CLK, etc), changing the source of a net, un-routing a net, manually routing a net (adding PIPs), and hundreds of other useful design transformations.

IV. Developing CAD Tools With RapidSmith

RapidSmith's capabilities are best illustrated through examples. This section presents several detailed coding examples that demonstrate how to create placers and routers with RapidSmith. In addition, this section will suggest other research areas where RapidSmith can be used to make contributions.

A. Placement

In order to create a placer, three basic capabilities are needed. First, an understanding of the objects or resources to be placed such as their connectivity, size, shape and classification is required. Second, a complete set of valid placement locations on which to place resources on the FPGA is needed. Finally, an infrastructure that allows a placer to quickly make small changes to a placement and evaluate its current state is an essential feature for many placement algorithms. RapidSmith provides all three of these capabilities and additionally includes frameworks to build graphical placement debugging tools to help in placer creation.

1) Instances: The basic building block found in an XDL design is the primitive instance. Instances are instantiations of a primitive type (such as a SLICEL, SLICEM, DSP48, etc.) and contain all logic and memory components of a design. Placement of instances is straightforward: simply assign the instance to a compatible primitive site. For example, consider the source code in Listing 1 which illustrates the basics of creating a new design, creating an instance, obtaining a list of valid placement sites and placing the instance in a site using RapidSmith. Note the method `getAllCompatibleSites()` which returns a complete list of all primitive sites for which a particular primitive type can be placed.

Listing 1: Simple Design Creation

```
// Create new design, set name and part
Design d = new Design("example", "xc5vlx30tff324");
PrimitiveType type = PrimitiveType.SLICEL;
Instance i = new Instance("foo", type);
d.addInstance(i); // Add the instance to the design
PrimitiveSite[] sites;
sites = d.getDevice().getAllCompatibleSites(type);
i.place(sites[0]); // Now, place the instance
```

2) Hard Macros, Modules and Module Instances: RapidSmith supports a core definition construct in XDL called a module. A module is a grouping of instances, nets and I/O connections called ports. Modules are useful in that they can represent hard macros (a circuit that was previously placed and routed and that can be placed as a single unit) in XDL and present a level of hierarchy in the design. A given module can be replicated multiple times in an XDL design. However, unlike conventional netlists, each module copy has its contents flattened into the XDL design (XDL designs are not truly hierarchical). These module contents, however, are tagged in a way that identifies from which module instantiation they came. To help preserve the hierarchy implied by the use of modules, RapidSmith introduces a `ModuleInstance` class that collects and encapsulates the flattened module contents into an object, facilitating their placement and manipulation.

A module instance may contain dozens (or even hundreds) of instances and routed nets that all must be checked when attempting placement. A module instance contains a specific instance that acts as a reference point to all other instances and nets in the module instance, this instance is called the anchor. Relative offsets are calculated from the anchor to ensure all the components of a module instance are placed in their correct relative positions. All of the complicated operations involved in placing module instances are encapsulated in just a few simple RapidSmith methods which are shown in Listing 2 to

create and place a module instance. Ultimately, these methods saved enough complexity that the hard macro placer used in [2] uses less than 350 lines of code.

Listing 2: Module Instance Creation and Placement

```
// Load an XDL file into RapidSmith
Design d = new Design("moduleContainingDesign.xdl");
// Get the 1024-FFT module definition by name
Module m = d.getModule("fft1024");
// Create an instance of the FFT module called "f0"
ModuleInstance mi = d.createModuleInstance("f0",m);
// Find all compatible sites with the anchor
PrimitiveType type = mi.getAnchor().getType();
PrimitiveSite[] sites;
sites = d.getDevice().getAllCompatibleSites(type);
int i = 0;
while(!mi.place(sites[i++], d.getDevice()))
  if(i >= sites.length)
    error(mi.getName()+" has no valid placement!");
```

3) A Hard Macro (Module Instance) Placer: Placers typically need to determine the degree of connectivity between primitive instances and modules so that highly-connected modules can be placed more closely together. The placer developed in [2] used code similar to that shown in Listing 3 which quickly computes the relative connectivity between modules.

Listing 3: Calculating Connectivity for Module Instance

```
for(Net net : design.getNets()){
  String n1=net.getSource().getModuleInstanceName();
  if(n1 == null) continue; //src is not hard macro
  InstanceBlock source = instanceBlockMap.get(n1);
  for(Pin pin : net.getPins()){
    if(pin.isOutPin()) continue; // skip the src
    String n2 = pin.getModuleInstanceName();
    if(n2 == null) continue; //snk is not hard macro
    InstanceBlock sink = instanceBlockMap.get(n2);
    source.connectionCount[sink.getIndex()]++;
    sink.connectionCount[source.getIndex()]++;
  }
}
```

Each module instance in the design is assigned an index and wrapped in a special class built specifically for the placer called an `InstanceBlock`. This special class contains an array that keeps track of its connection count to every other module instance in the design. The code in Listing 3 makes a single pass over all the nets in order to tabulate connection counts between all module instances. This information is calculated once during initialization of the placer and used throughout the placement process to find a good placement.

4) An Interactive Hard Macro Placer: A good visualizer tool can be a great help when debugging and optimizing a placer. A visualizer for viewing hard macro placement was developed early on in our hard macro project. It proved to be invaluable for debugging and optimizing our hard macro placer. The device and design packages made it simple to draw an FPGA layout and corresponding hard macro shapes as pictured in Figure 3. The graphical tool also allows the user to interactively move the module instances around the chip and receive placement feedback in the form of colors to locate valid placements. Ultimately, the productivity provided by the manual hard macro placer was quite valuable in accelerating

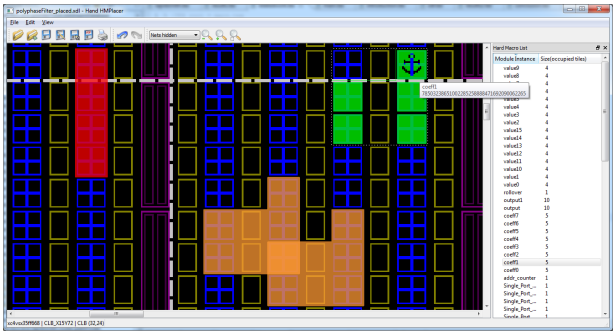


Fig. 3: Screen shot of interactive hard macro placer in RapidSmith; shows a hard macro illegally placed (red/top left), overlapping hard macros (orange/bottom), and an expanded hard macro to show utilized tiles (fragmented green/top right).

development. The tool also illustrates the customization capable with RapidSmith as the interactive placer has no other corresponding tool (Xilinx's FPGA Editor is unable to provide interactive real-time feedback for placing hard macros).

B. Routing

To create a router for Xilinx devices, a user must have a reasonable understanding of (1) the routing graph available in Xilinx parts, (2) wire connection types and representation, (3) power and ground distribution issues, and (4) various architectural nuances. RapidSmith aids the user in many of the difficult aspects of writing a router by providing a compact and efficient routing graph, significant functionality through its APIs and framework, and examples in order to make writing a custom router feasible.

1) RapidSmith Routing Graph: A wire, as presented in XDLRC reports, has a name which is unique in the tile where it resides. However, multiple copies of a wire with the same name can be found in several different tiles due to the regular replicated structure of Xilinx FPGAs. Therefore, to reference a specific routing resource on a device, both the wire name and tile are needed to eliminate ambiguity.

In order to provide a small, yet, high performance routing graph, RapidSmith uses an efficient technique to store and retrieve routing resources (nodes) and routing connections (edges). RapidSmith represents specific routing resources in the `Node` class which contains a tile and wire reference (stored as an `int`) as well as other routing aids such as a cost variable and parent reference.

Routing connections, or edges in the routing graph are represented in the `WireConnection` class in conjunction with a hash map found in each tile. Each wire in a tile is a key in the tile's hash map and the key's corresponding value is an array of `WireConnection` instances. The `WireConnection` class contains a wire and relative offsets from the key wire's tile. RapidSmith uses a relative set of tile offsets instead of absolute offsets for the memory savings gained by reusing identical `WireConnection` instances all throughout the device.

In order to traverse the routing graph, `Node` instances are created for each routing resource visited in the graph. This is

readily apparent in Listing 4 which shows a very basic method of routing a source to a sink.

Listing 4: Basic Routing Method

```
public ArrayList<Node> route(Pin src, Pin snk){
 Device dev = Device.getInstance("xc4vfx12ff668");
 PriorityQueue<Node> pq=new PriorityQueue<Node>();
 Node snkNode = dev.getNodeFromPin(snk);
 Node currNode = dev.getNodeFromPin(src);
 // Loop on queue output nodes to find the sink
 while(!currNode.equals(snkNode)){
   WireConnection[] conns=currNode.getConnections();
   if(conns != null)
     for(WireConnection w : conns){
       // create a new node, n.parent=currNode
       Node n = w.createNode(currNode);
       n.setCost(n.getManhattanDistance(snkNode));
       if(!pq.contains(n)) pq.add(n);
     }
   if(pq.isEmpty()) return null;
   currNode = pq.remove();
 }
 // When we have found the sink, reconstruct path
 ArrayList<Node> path = new ArrayList<Node>();
 while(currNode.getParent() != null){
   path.add(currNode);
   currNode = currNode.getParent();
 }
 return path;
}
```

The method starts by creating nodes for the source and sink pins as the wire names used in pins require mapping to PIP wire names. This pin mapping was previously identified in Section III as one of the areas where XDLRC reports fall short of providing sufficient information to make all necessary mappings. As can be seen from Listing 4, RapidSmith provides methods to handle the mapping automatically.

A priority queue orders nodes for processing by the router. The cost function used to prioritize the nodes is a very simple Manhattan distance of the current node's tile to the sink node's tile. The algorithm continues to poll the queue looking for new connections and adding them to the queue until the sink node is found. Once the sink node is found, the parent references of the nodes are followed back to the original source to create the routed path. A separate method (not shown) could easily convert the list of nodes into PIPs for the final routed net.

2) Power/Ground Net Accommodations in RapidSmith: Power and ground net routing is handled in two steps. First, an un-routed design contains one net which contains all ground sink pins and one net which contains all power sink pins. These two nets must be partitioned into local neighborhood nets based on switch box power/ground posts called TIEOFFs. Second, although TIEOFFs are conveniently located in each switch matrix, there often exist input pins that require power or ground that cannot be supplied by the TIEOFFs in certain cases or due to routing congestion. In those situations, LUTs are configured nearby to provide the needed source.

RapidSmith provides functionality in the form of the `StaticSourceHandler` class that automates this two step process of partitioning ground and power nets and creating LUTs when needed. Because this functionality is included in RapidSmith, prospective router writers do not have to implement the tedious ground and power distribution process, but can focus on their unique algorithm at hand.

3) Design Analysis for Router Congestion Avoidance: One example which demonstrates the flexibility of RapidSmith in router construction can be found in our previous work [2]. Our attempt to write a very fast router prompted us to abandon the traditional PathFinder algorithm [6] for a simple maze router. Since a maze router only makes a single pass over the nets of a design, it has no mechanisms to deal with congestion. This can cause the router to fail when areas of a chip are heavily congested and nets cannot be routed due to a lack of coordinated resource allocation.

Because RapidSmith is able to provide a flexible platform for creating the router, we were able to introduce the idea of reserving specific routing resources for nets which could be threatened by congestion. The congestion avoidance techniques were highly architecture specific but provided insights into the routing structure that helped us understand which resources were often in demand.

C. Applying RapidSmith to Other Areas of FPGA Research

1) Post-PAR Design Analysis: RapidSmith provides support for the creation of a variety of powerful custom post-PAR analysis tools not possible using other means. One such example arises in FPGA reliability where high energy particle strikes can upset a design's behavior. Known techniques such as circuit triplication with voters and bitstream scrubbing can mitigate against particle strikes in the bitstream. However, half-latches (weak keepers) inserted by the Xilinx flow to prevent unspecified signals from floating (such as clock enable signals to flip flops) [7], are still vulnerable to such strikes as half-latches are not controlled by the bitstream. Since the user has no control over half-latch insertion into circuits produced by the normal PAR process, the identification of half-latches and their removal must be done on post-PAR designs.

The detection of half-latches on flip flop clock enable wires using RapidSmith is trivial and is shown in Listing 5. The basic steps are: (1) loop across all instances in the design and identify the slices, (2) for each such slice, see if the slice contains configured flip flops (by examining the instance's "FFX" and "FFY" attributes), (3) for all such slices, examine the slice's "CEUSED" attribute — if its value is "OFF", then a half-latch exists in the slice.

Listing 5: Half-Latch Detection

```
for(Instance i : design.getInstances())
  if(i.getType().equals(PrimitiveType.SLICEL) ||
     i.getType().equals(PrimitiveType.SLICEM))
    if(i.testAttributeValue("FFX", "FF") ||
       i.testAttributeValue("FFY", "FF"))
      if(i.testAttributeValue("CEUSED", "OFF"))
        System.out.println("Instance: "+i.getName()
            + " has a half-latch on CE pin");
```

2) FPGA Security: A growing area of FPGA research is the vulnerability of FPGAs to hackers and security holes. RapidSmith enables very detailed experiments that allows researchers to conduct "what-if" experiments to see how

difficult it might be to perform a certain kind of attack. For example, Tavaragiri et al. [8] demonstrate (using specialized XDL tools) usable antennas constructed from the configurable routing on a device to radiate information off-chip. Although RapidSmith was not used for these experiments, RapidSmith is capable of creating the same kinds of structures in designs.

Others have demonstrated the ability to modulate the power rail voltage in FPGAs in order to communicate information outside the device [9]. Performing side-channel attacks require very specialized FPGA design structures which can only be created with very careful, low level manipulation. In general, RapidSmith opens up new research avenues by providing support that makes it feasible to create a variety of special-purpose tools that go well beyond the typical placer or router.

V. Related Work

The Xilinx Design Language is not new. XDL and the accompanying `xdl` tool in the Xilinx ISE design suite have been present for over 10 years [10]. Others have also realized the potential of XDL as several papers have been published that demonstrate its use in ideas such as a bus macro generator [11], C-slow re-timing [12], power estimation [13], floor planning tools [14], routing constraints [15][16] and run-time reconfiguration [17]. Despite these and several other efforts which leverage XDL, a unified open source solution to facilitate the use of XDL has never materialized until recently. Steiner [18] and Kepa et al. [19] both demonstrate functional XDLRC-derived device databases, however they have not (at the time of writing) been openly released.

Recently, our colleagues at USC-ISI have released Torc [20], an open source tool that shares the vision of doing research on commercial FPGAs. RapidSmith and Torc differ in target audience (Torc is written in C++ while RapidSmith is in Java), but do have similar functionality in that Torc also leverages XDL and builds device databases that enable CAD tool creation. Torc has been used as the foundation of OpenPR [15] an open source partial reconfiguration tool.

VI. Conclusion

In this work we have introduced RapidSmith, an open source platform for rapidly creating FPGA CAD tools. RapidSmith is written in Java and provides FPGA researchers with a common platform to implement new ideas and algorithms on Xilinx FPGAs. We have also illustrated the successful implementation of placers and routers using RapidSmith as well as its use in several other areas of FPGA research.

RapidSmith supports all modern Xilinx devices: Virtex{E, 2, 2-Pro, 4, 5, 6, 7}, Spartan{2, 2E, 3, 3A, 3ADSP, 3E, 6} and Kintex 7 families. RapidSmith includes several examples, Javadocs, documentation and source code to get users started and can be downloaded at http://rapidsmith.sf.net.

References

[1] V. Betz and J. Rose, "VPR: A New Packing, Placement And Routing Tool For FPGA Research," in *Proceedings of the 7th International Workshop on Field-Programmable Logic and Applications*. Springer-Verlag London, UK, 1997, pp. 213–222.

[2] C. Lavin, M. Padilla, J. Lamprecht, P. Lundrigan, B. Nelson, and B. Hutchings, "HMFlow: Accelerating FPGA Compilation with Hard Macros for Rapid Prototyping," in *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on*, May 2011, pp. 117–124.

[3] C. Lavin, M. Padilla, P. Lundrigan, B. Nelson, and B. Hutchings, "Rapid Prototyping Tools for FPGA Designs: RapidSmith," in *Field-Programmable Technology (FPT'10). International Conference on*, December 2010.

[4] S. Malhotra, T. Borer, D. Singh, and S. Brown, "The Quartus University Interface Program: enabling advanced FPGA research," in *Field-Programmable Technology, 2004. Proceedings. 2004 IEEE International Conference on*, December 2004, pp. 225–230.

[5] S. Ferguson and E. Ong, "Hessian 2.0 Serialization Protocol," http://hessian.caucho.com/doc/hessian-serialization.html, August 2007.

[6] L. McMurchie and C. Ebeling, "PathFinder: a Negotiation-based Performance-driven Router for FPGAs," in *Proceedings of the 1995 ACM Third International Symposium on Field-programmable Gate Arrays*, ser. FPGA '95. New York, NY, USA: ACM, 1995, pp. 111–117.

[7] H. Quinn, G. Allen, G. Swift, C. W. Tseng, P. Graham, K. Morgan, and P. Ostler, "SEU-Susceptibility of Logical Constants in Xilinx FPGA Designs," *Nuclear Science, IEEE Transactions on*, vol. 56, no. 6, pp. 3527 –3533, December 2009.

[8] A. Tavaragiri, J. Couch, and P. Athanas, "Exploration of FPGA Interconnect for the Design of Unconventional Antennas," in *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, ser. FPGA '11, 2011, pp. 219–226.

[9] D. Ziener, F. Baueregger, and J. Teich, "Using the Power Side Channel of FPGAs for Communication," in *Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on*, May 2010, pp. 237 –244.

[10] *Xilinx Design Language Version 1.6*, Xilinx, Inc., Xilinx ISE 6.1i Documentation in ise6.1i/help/data/xdl, July 2000.

[11] C. Claus, B. Zhang, M. Huebner, C. Schmutzler, J. Becker, and W. Stechele, "An XDL-based Busmacro Generator for Customizable Communication Interfaces for Dynamically and Partially Reconfigurable Systems," in *Workshop on Reconfigurable Computing Education at ISVLSI 2007*, Porto Alegre, Brazil, May 2007.

[12] N. Weaver, Y. Markovskiy, Y. Patel, and J. Wawrzynek, "Post-placement C-slow Retiming for the Xilinx Virtex FPGA," in *Proceedings of the 2003 ACM/SIGDA Eleventh International Symposium on Field Programmable Gate Arrays*, ser. FPGA '03, 2003, pp. 185–194.

[13] V. Degalahal and T. Tuan, "Methodology for High Level Estimation of FPGA Power Consumption," in *Design Automation Conference, 2005. Proceedings of the ASP-DAC 2005. Asia and South Pacific*, vol. 1, January 2005, pp. 657 – 660 Vol. 1.

[14] D. Koch, C. Beckhoff, and J. Teich, "ReCoBus-Builder A Novel Tool and Technique to Build Statically and Dynamically Reconfigurable Systems for FPGAs," in *Field Programmable Logic and Applications (FPL'08). International Conference on*, September 2008, pp. 119–124.

[15] A. Sohanghpurwala, P. Athanas, T. Frangieh, and A. Wood, "OpenPR: An Open-Source Partial-Reconfiguration Toolkit for Xilinx FPGAs," in *18th Reconfigurable Architectures Workshop (RAW 2011)*, May 2011.

[16] D. Koch and J. Torresen, "Routing Optimizations for Component-based System Design and Partial Run-time Reconfiguration on FPGAs," in *Field-Programmable Technology (FPT'10). International Conference on*, December 2010.

[17] K. Puttegowda, W. Worek, N. Pappas, A. Dandapani, P. Athanas, and A. Dickerman, "A Run-time Reconfigurable System for Gene-sequence Searching," in *VLSI Design, 2003. Proceedings. 16th International Conference on*, January 2003, pp. 561 – 566.

[18] N. Steiner, "A Standalone Wire Database for Routing and Tracing in Xilinx Virtex, Virtex-E, and Virtex-II FPGAs," Master's thesis, Virginia Polytechnic Institute and State University, 2002.

[19] K. Kepa, F. Morgan, K. Kosciuszkiewicz, L. Braun, M. Hubner, and J. Becker, "FPGA Analysis Tool: High-Level Flows for Low-Level Design Analysis in Reconfigurable Computing," in *Proceedings of the 5th International Workshop on Reconfigurable Computing: Architectures, Tools and Applications*, ser. ARC '09, 2009, pp. 62–73.

[20] N. Steiner, A. Wood, H. Shojaei, J. Couch, P. Athanas, and M. French, "Torc: Towards an Open-Source Tool Flow," in *Proceedings of the 19th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '11. New York, NY, USA: ACM, 2011.