RapidWright: Enabling Custom Crafted Implementations for FPGAs

Chris Lavin and Alireza Kaviani
Xilinx Research Labs
San Jose, CA, USA
{chris.lavin,alireza.kaviani}@xilinx.com

***Abstract*—FPGA application size is rapidly growing by reuse and replication. Achieved quality of results (QoR) of these large designs is often much lower than what could be realized with localized circuits at a modular level. One underlying reason for QoR loss is that back-end implementation tools compile the designs as one large circuit entry. Is there a way to bring innovation to the implementation stage of FPGA compilation that can improve QoR?**

This work proposes a pre-implemented methodology for FPGAs to achieve higher performance or productivity and introduces RapidWright, an open-source platform to enable this new approach. We aim to enhance either QoR or productivity through the reuse of modular implementations and present examples that improve QoR up to 50% or accelerate compilation time and debug by more than an order of magnitude. Finally, we demonstrate how RapidWright enables custom crafted implementations with near spec performance.

***Keywords*-FPGA; Xilinx; Pre-implemented; QoR; Debug; Clocking; Overlays; Shells;**

I. Introduction

FPGA implementation tools lead a challenging life. On one hand, they must keep pace with FPGAs that have grown rapidly in both capacity and performance. On the other hand, FPGA growth has enabled significant advances in both integration and functionality, further magnifying the support burden of the tools. As FPGAs are designed to support a wider range of applications than any ASIC or ASSP counterpart, the tools must accommodate a broad design space by synthesizing and implementing circuits of many domains. Thus, FPGA tools, such as Xilinx's Vivado, have become highly complex software systems that must perform this feat for all devices in the vendor's product portfolio.

The necessity of breadth coverage by commercial tools often leads to implementations that do not take full advantage of the underlying hardware. For example, UltraScale+ devices employ DSP blocks that are rated at 891MHz for the fastest speed grade. Nonetheless, large designs implemented on FPGAs typically achieve system frequencies lower than 400MHz. In this work, we propose a methodology with the goal of delivering near-spec performance, addressing this performance gap.

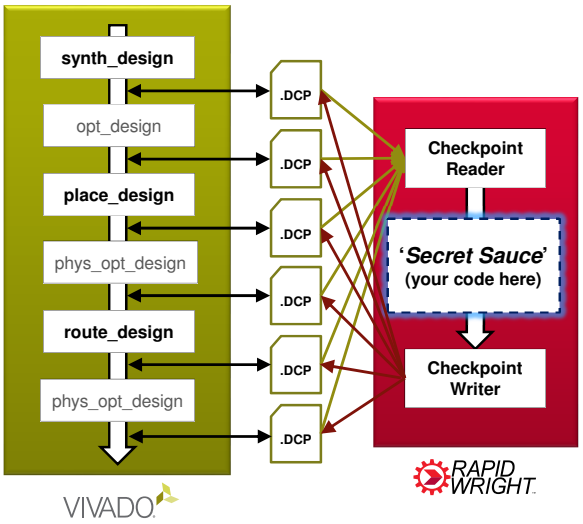


Figure 1. Vivado and RapidWright DCP Compatibility

We introduce *RapidWright*[1], an open source platform that provides a gateway to Vivado's back-end implementation tools (see Figure 1) in order to realize the full potential of advanced FPGA silicon. RapidWright works synergistically with Vivado to produce highly-tuned, custom implementations for emerging applications. It builds on the premise of two key observations: (1) Vendor tools such as Vivado often produce high performance results for small modules of a design. (2) Emerging applications such as compute-bound or machine learning designs grow in size by module replication. We rely on Vivado to produce highly optimized implementations for key modules of a design to deliver the highest performance. RapidWright can then replicate, relocate and assemble these tuned modules to compose a complete application while maintaining high performance.

The RapidWright platform employs three key capabilities to sustain global system performance. First, it preserves high quality placement and routing of pre-implemented blocks. Second, it enables reuse and replication of blocks to leverage Vivado's efforts of achieving high quality results. And third, RapidWright stitches the blocks together with minimal or no loss of quality. Leveraging these capabilities, we demonstrate how to augment Vivado and close the performance gap between device

[1]Wright = maker or builder

data sheets and achievable results through vendor tools. Keep in mind, relocating pre-implemented modules is a challenging task and not feasible in Vivado in all but the smallest cases. RapidWright ultimately enables a new mode of implementation composition that has not been feasible previously and this work demonstrates how these capabilities contribute to both performance and productivity. Specific contributions of this work include:

1) An open source platform that enables crafting customized module-based implementations of FPGA applications.
2) Specific case studies showing how RapidWright improves design performance and productivity.
3) A versatile gateway to Vivado by reading and writing design checkpoints (DCPs) natively — setting the ground work of an academic ecosystem for further advancing FPGA tools.

The third contribution empowers researchers by combining the commercial credibility of FPGA tools with the agility of an open source framework, leading to innovative solutions that might not be feasible otherwise. The next section summarizes background and related work. Section III describes core RapidWright structure. Section IV proposes a pre-implemented design strategy built into RapidWright followed by a number of example use cases and results in Section V. We conclude and outline future work in Section VI.

II. Related Work

There have been a number of open interfaces to FPGA vendor tools in the past. The Xilinx Design Language (XDL) [1] provided an open design file format released with ISE (predecessor to Vivado). XDL provided unprecedented access to placement and routing information that made building custom CAD tools possible. Open source projects and tools such as RapidSmith [2], Torc [3], and ReCoBus-Builder [4] capitalized on the XDL interface to increase productivity. Altera (now part of Intel) also released the Quartus University Interface Program (QUIP) [5] with similar attributes to XDL.

XDL and QUIP are no longer supported by the latest vendor tool suites Vivado and Quartus Prime. Instead, Xilinx's Vivado utilizes a new Tcl-based interpreter. This allows users to perform customized tasks by writing Tcl scripts. For small tasks, this is not a problem. However, as task size and complexity grows, productivity is limited by Tcl interpreter performance. For example, Tincr [6] and the Vivado Design Interface (VDI) [7] have extended RapidSmith to be compatible with Vivado by replacing XDL with Tcl interface routines. However, design import by this method is constrained by the low speed of Tcl commands, limiting feasibility to only the smallest designs. Townsend [7] reports a design of ~54K LUTs targeting an xc7a100t (Artix 7) takes ~2.5 hours to import into Vivado (6 LUTs/second). By rough comparison, RapidWright writes a 210K LUT DCP targeting an xcvu190 (Virtex UltraScale) in 91 seconds. Vivado reads the DCP in 303 seconds, for a total import time of 394 seconds, or 533 LUTs/second, 88× faster than VDI. By avoiding Tcl and using DCPs, RapidWright enables a more productive interface.

III. RapidWright: Foundation and Structure

To support RapidWright's mission of empowering users to craft customized implementations within an open source framework, three pieces of infrastructure are needed. First, an accurate device model of the underlying architecture. Second, logical and physical design models with representative data structures and APIs. Lastly, readers and writers for DCP file components that populate and export to/from design models—providing a gateway to Vivado.

A. The Device Model

The device model is compiled from a detailed textual report derived from externally available device data from Vivado. RapidWright parses this report, called a Xilinx Device Description (XDD) file, for each device and generates an internal device database file as shown in Figure 2. RapidWright faithfully reproduces nearly all device objects present within Vivado [8], including logic constructs such as BELs (basic elements of logic), sites, tiles, clock regions (CRs) and super logic regions (SLRs). It also includes all routing constructs such as site wires, BEL pins, site pips, site pins, wires, pips, and nodes.

The performance of a select number of device models is shown in Table I. As seen in the table, even the largest Xilinx device loads in <2 seconds and consumes only about 400MBs. These load times are an order of magnitude faster than a corresponding load by Vivado.

B. The Design Model

Two major components are central to the design model within Vivado and RapidWright. First, in contrast to

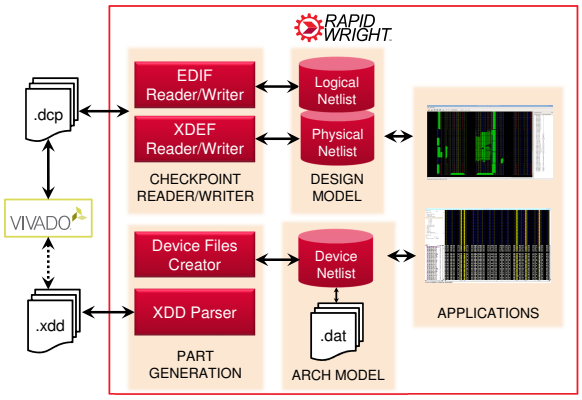


Figure 2. RapidWright Infrastructure Components

Table I
SELECTED RAPIDWRIGHT DEVICE LOAD CHARACTERISTICS[a]

Part	Load Time	Memory	LUTs	Platform
xc7a12t	0.21s	53MB	8,000	
xc7z020	0.21s	74MB	53,200	PYNQ-Z1
xcku040	0.91s	189MB	242,400	KCU105
xczu19eg	1.85s	386MB	522,720	
xc7v2000t	1.86s	339MB	1,221,600	
xcvu9p	1.97s	409MB	1,182,240	AWS F1
xcvu440	1.98s	393MB	2,532,960	

[a]Average of 10 runs on an Intel Core i7-4600U 2.10GHz, 16GB RAM and 250GB SSD using Oracle JRE 1.8.0_45 JVM.

the previous generation ISE tools, Vivado preserves and maintains the logical netlist throughout the entire implementation flow. Second, a physical netlist is formed during implementation. A physical netlist maps primitive cells within the logical netlist to BEL sites on the device and assigns nets to configurable routing interconnect resources. RapidWright represents both logical and physical netlists through data models of hierarchical classes and APIs.

1) Logical Netlist: RapidWright capitalizes on Vivado's support of EDIF [9] (in DCP and Tcl APIs) by using it as the logical netlist exchange format and provides an EDIF reader, writer and logical netlist data structures. An unencrypted EDIF netlist is a prerequisite to open any design in RapidWright as Vivado uses netlist name compatibility for placement and routing information. RapidWright contains various APIs to query, traverse and modify the logical netlist and provides direct mappings to the physical netlist.

2) Physical Netlist: A core portion of RapidWright is the physical netlist model. The physical netlist is responsible for modeling the placement and routing information stored in a DCP and makes it accessible to the user. Although the specific details and representations provided in RapidWright are beyond the scope of this section, there are adequate facilities within the framework to perform customized operations such as placement, routing, application-specific clocking, statistical analysis and module relocation and replication.

C. Design Checkpoint Readers and Writers

An enabling feature of RapidWright is how it directly reads/writes design checkpoint (DCP) files from/to Vivado and populates the design model accordingly. A DCP is a Vivado file that represents a design snapshot at any stage of the design/implementation process. A DCP file is actually a `.ZIP` file with a `.DCP` extension and multiple files are stored inside. The logical netlist of a design is an EDIF [9] file and the physical netlist is stored in an XDEF (internal Xilinx binary format) file. As shown in Figure 2, the RapidWright framework includes EDIF and XDEF readers and writers that have been created specifically to support DCP files. These readers and writers are fully tested with hundreds of different designs across multiple architecture by validating round trip accuracy using placement and routing reporting methods in Vivado. RapidWright also preserves and minimally parses other essential DCP files (such as constraints) to ensure full design reproducibility.

IV. A PRE-IMPLEMENTED MODULAR DESIGN STRATEGY

One of the key attributes of RapidWright is the ability to capture optimized placement and routing solutions for a module and reuse them in multiple contexts or locations on a device. Vivado often provides good results for small implementation problems (smaller than 10k LUTs within a clock region). However, when those same modules are combined into a large system, total compile time increases and the probability of timing closure is reduced. This phenomenon limits achievable performance and timing closure predictability of larger designs. We show how to preserve and reuse high quality solutions in RapidWright with pre-implemented modules, and propose a methodology of how they can improve the overall system performance in a large design.

A. Pre-implemented Modules

Pre-implemented modules are self-contained netlist cells that contain relative placement and routing information (generally with a rectangular footprint) targeting a specific FPGA device. RapidWright generates pre-implemented modules by invoking Vivado to synthesize, place and route them out-of-context (OOC) of the original design. RapidWright then preserves and packages the placement and routing information from the OOC DCP.

For a pre-implemented module to be reusable, it often needs to be area constrained with a pblock containing the attribute `CONTAIN_ROUTING=1`. This ensures that placement and routing of the module is restricted to the respective rectangle, reducing its footprint such that it has a higher number of compatible placement locations across the device.

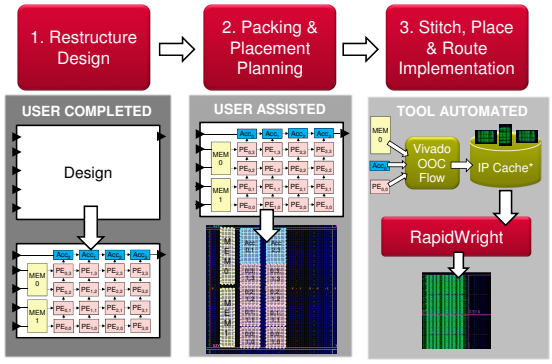


Figure 3. RapidWright Pre-implemented Design Strategy

B. Design Strategy and Flow

RapidWright endows users with a new design vocabulary by caching, reusing and relocating pre-implemented blocks. We believe this to be an enabling concept and offer a high performance design strategy as depicted in Figure 3.

The first step requires the design architect to restructure the proposed design such that it can take full advantage of the benefits provided by pre-implemented modules. We define restructuring as a design refactoring that reflects three favorable design characteristics: (1) modularity, (2) module replication and (3) latency tolerance. Modularity uncovers design structure so it can be strategically mapped to architectural patterns. When modules are replicated, reuse of those high quality solutions and architectural patterns can be exploited to increase the benefits. Finally, if the modules within a design tolerate additional latency, inserting pipeline elements between them improves both timing performance and relocatability.

After the design architect has successfully restructured and modularized a design, step two of Figure 3 is followed. Here, the design architect creates an implementation guide file that captures how best to map the modules of a design to the architecture of the target device. Specifically, pblocks are chosen for those pre-implemented modules of interest and physical locations are chosen for each instance. This step provides the design architect an opportunity to navigate FPGA fabric discontinuities. These discontinuities include boundaries such as IO columns, processor subsystems, and most significantly, SLR crossings. Such architectural obstacles cause design disruptions when targeting high performance. However, by leveraging the pre-implemented methodology provided in RapidWright, custom-created implementation solutions can be identified and planned out to manage the fabric discontinuities by custom module placement. Ultimately, this process is iterative and can inform useful RTL/design changes by focusing design structure to better match architectural resources.

Step three of the design strategy is an automated flow provided with RapidWright, whose details are denoted in Figure 4. We leverage a design input method in Vivado called IP Integrator (IPI)[10]. IPI offers an interactive block-based approach for system design by providing an IP library, IP creation flow and IP caching. RapidWright takes advantage of IPI by using leaf IP blocks as de-facto pre-implemented blocks and also by leveraging the IP caching mechanism. The RapidWright pre-implemented flow extends the caching mechanism to go beyond synthesis, by performing OOC placement and routing on the block within a constrained area. The flow begins by invoking Vivado's typical IPI synthesis and creating pre-

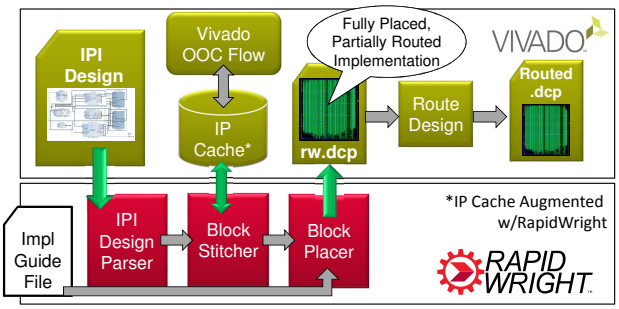


Figure 4. Pre-implemented Block Compilation Flow

implemented blocks for each module if not already found in the cache. RapidWright has an IPI Design Parser (EDIF-based) that creates a black-box netlist where each instance of a module is empty, ready to receive the pre-implemented module guts. The block stitcher reads the IP cache and populates the IPI design netlist. After stitching, the blocks are placed according to the implementation guide file from the design architect. Once all the blocks are placed, RapidWright creates a DCP file that is read into Vivado that completes the final routes.

V. Results: RapidWright Deployment Examples

This section details three use cases that exemplify the performance and productivity gains possible by leveraging RapidWright customization capabilities. First, we show how to improve performance by leveraging the pre-implemented strategy and flow described in Section IV. In the second subsection we focus on design productivity and how pre-implemented modules can benefit both compile time and debug. Finally, we demonstrate near-spec performance on an AWS F1 instance design by combining the benefits of our pre-implemented strategy and customization capabilities of RapidWright.

A. Capturing Performance by Pre-implemented Construction

We present four designs on which we applied the pre-implemented strategy and corresponding performance results can be seen in Table II. These designs were taken from scenarios where the underlying source was already optimized (baseline). To help quantify the benefits of the proposed strategy, results are reported for the restructured design using both conventional Vivado and the pre-implemented approach.

The SEISMIC design includes 660 identical modules chained by input and output dependencies on its three previous and three next neighbors. Each module is a seismic data computation and consumes 252 LUTs and 342 flops after restructuring, with the entire implementation consuming 93% of all device LUTs. After experimentation of different pblocks, it was found that a 3x15 CLE

Table II
PRE-IMPLEMENTED MODULE FLOW RESULTS

Design	Device	LUTs	FFs	DSPs	BRAMs	Baseline	Restructured	Δ	Pre-implemented	Total Δ
SEISMIC	KU040	93%	5%	-	-	270MHz	354MHz	31%	390MHz	41%
FMA	KU115	25%	50%	97%	6%	270MHz	273MHz	1%	417MHz	54%
SGEMM	KU115*	19%	20%	87%	-	391MHz	437MHz	10%	462MHz	16%
ML	ZU9EG*	46%	29%	42%	96%	368MHz	569MHz	55%	541MHz	50%

*Constrained portion of the device

area constraint (integer multiple of CR height) produced the best performance and fabric usage efficiency.

As the connectivity of the SEISMIC modules was a linear nearest neighbor progression, some experiments of placement patterns were run and partial screenshots of module layouts (per a RapidWright module exploration tool) are shown in Figure 5. Although three separate packing patterns were attempted, all yielded f_{max} figures within ±3% demonstrating that the fabric interconnect flexibility was not the underlying bottleneck in the design.

The FMA design is another benchmark that has 1415 modules, of which 1340 are 16-bit multiply accumulate (MAC) operations. It also includes an SDAccel shell design, allowing communication to a host over PCIExpress. The MAC modules consumed 4 DSP48E2s each and through experimentation, it was found that a 4x10 CLE pblock (including 4 DSPs) yielded the best performance. The MAC modules connect in a nearest neighbor linear pipeline and a serpentine placement pattern yielded the best performance (similar to that used in Figure 5a). Of significance, the MAC modules communicated with an AXI streaming bus protocol that contained a ready feedback signal between each block that often became the critical path in our experiments.

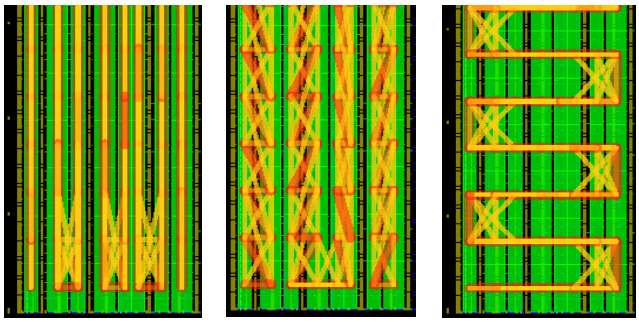
The FMA implementation dealt with several fabric discontinuities: inconsistent tile patterns, IO columns and SLR crossings. A screenshot of the bottom half (lower SLR) of the FMA design can be seen in Figure 6. Six different pre-implemented versions of the MAC module were created to get maximum utilization of the FPGA fabric. Due to fabric discontinuities, additional implementations (1, 2 and 3) are necessary to provide coverage for all CLB/DSP column patterns. Implementations 4 and 5 addressed the depopulated CLBs for SLR crossing tiles, but only 4 instances instead of 6 fit due to fewer CLBs.

As the MAC serpentine chain wound its way around the chip, it had to cross IO columns (black vertical bars in Figure 6), additional pipeline elements were placed at the edges to ensure they (and the ready feedback signal) did not become a bottleneck. Ultimately, we achieved a 54% improvement over unaided Vivado (baseline) with the final critical path being signals crossing the SLR.

The SGEMM design performs a single precision floating point general matrix multiplication and uses 150 identical modules to complete the operation. Unfortunately, the 16 DSP48E2 primitives needed per module created a poor architectural mapping to the 24 in a CR DSP column. Due to this mismatch, the SGEMM design only had minor improvements using the pre-implemented module flow.

Our final benchmark design, ML, is a machine learning application that had several blocks types, of which the majority were processing cells (16x16 array). This design underwent several changes in order to make it amenable to IPI and the pre-implemented blocks flow. In fact, the restructured design compiled in Vivado produced a slightly better result than the pre-implemented module flow. The lower than expected performance from the pre-implemented flow was caused by an inter-block routing issue we discuss further at the conclusion of this section.

The key takeaway from this subsection is two-fold. The first two designs highlight how an application ar-



(a) Vertical serpentine (b) Vertical zig-zag (c) Horizontal serpentine

Figure 5. SEISMIC Module Placement Patterns Created with RapidWright

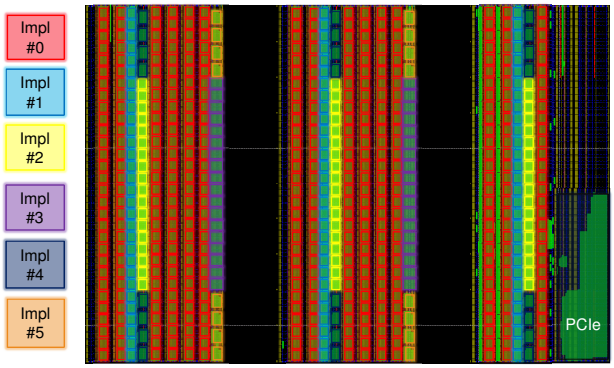


Figure 6. FMA Design Module Layout in Bottom SLR

chitect can leverage the proposed strategy and flow to explore a variety of module placements. The FMA design success is enabled by proactive module placement to compensate for fabric discontinuities. In contrast, the last two designs show that such success is not pervasive and highly depends on the nature of the design. We also observe that the refactoring process is independently beneficial as in the ML design example. In the end, our methodology is more likely to succeed with larger designs of high replication with full awareness of the target architecture.

B. Productivity by Modular Reuse

As design sizes grow, compile times increase and productivity is adversely affected. To help alleviate this trend, we demonstrate how RapidWright can leverage module reuse to reduce both compile time and implementation iterations for debug scenarios.

One way to reduce compile time, is to reuse existing results. For example, if the IP cache mentioned in Section V-A is fully populated with pre-implemented modules and the flow uses the automated placement and routing tools within RapidWright, fast implementations can be achieved, enabling approaches such as HMFlow [11] and TFlow [12] for Vivado.

As a point of reference, an example IPI MicroBlaze design included with Vivado can be compiled in 232 seconds. In contrast, the same design compiles in 12.5 seconds in RapidWright with a full cache, almost 20× faster at the expense of lower achieved f_{max}. Such productivity increase can prove quite useful for emulation applications with less stringent performance requirements. RapidWright includes all the components of this flow and can easily be called with a single Tcl procedure in Vivado/IPI. Thus, users will easily be able to run both conventional and fast flows from the same IPI design depending on their specific needs and use case.

Another way of improving productivity is through faster debug compiles. Debugging on FPGAs, without significant planning, has typically required expensive recompiles of the design. Unfortunately, recompiling a design exposes the user to additional unpredictability that manifests in frustrating ways such as hiding a bug, additional bugs, or other misunderstood behavior. Previous work has demonstrated feasibility of post implementation instrumentation [13][14] and this work applies these techniques with commercial debugging tools.

Vivado provides an Integrated Logic Analyzer (ILA– previously called ChipScope) that is inserted into a netlist before place and route to provide the user visibility during runtime of the circuit. Using RapidWright, we demonstrate a debug flow that adds an ILA and probe routing to a design without disturbing existing placed and routed logic.

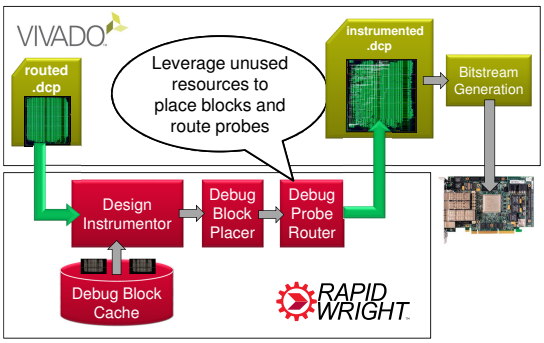


Figure 7. Post-route Debug Instrumentation Flow

Table III
Post Implementation Debug Flow Results

Design	CLB	BRAM	DSP	Vivado Baseline	This Work	Δ
dsp1	9%	0%	38%	1455s	42s	35×
10g	10%	6%	<1%	260s	11s	24×
dsp2	20%	0%	78%	1057s	89s	12×
sparc	31%	10%	<1%	973s	10s	97×
21ch	70%	93%	18%	1660s	50s	33×

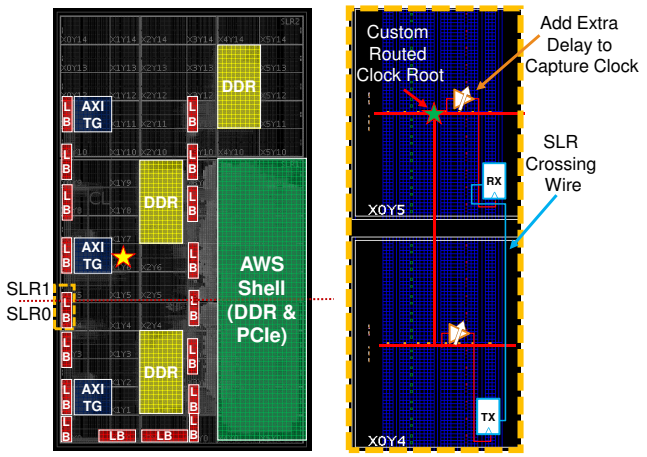
The flow in Figure 7 reads a placed and routed DCP into the RapidWright design instrumentor that identifies nets marked for debug (previously annotated in Vivado) and checks the cache for compatible ILAs. If a suitable ILA instance is not found, RapidWright invokes Vivado to place and route a compatible one and adds it to the cache. Once a pre-implemented ILA is secured, it is stitched directly into the top level netlist of the design and probes are attached to the appropriate nets. After ILA insertion, it is placed by the debug block placer. Routing is quickly accomplished by a rudimentary router provided in RapidWright that preserves existing routes and only consumes unused resources.

We tested this flow with five different designs of varying size and composition. All designs targeted the xcku040-ffva1156-2-e (Kintex UltraScale) and used Vivado 2016.1. Our baseline runtime comparison is the sum total of Vivado's `opt_design` (which adds the debug instrumentation logic), `place_design` and `route_design` functions. To measure runtime for our flow, the sum total runtime of the design instrumentor, Debug Block Placer and Debug Probe Router is reported in Table III.

From Table III, it can been seen that the device utilization ranged from 9% to 70% of CLBs and speedup achieved ranged from 12-97×. Note that we report CLB counts instead of LUTs as we do not attempt to partially use CLBs. Selected designs were downloaded to a KCU105 board and used in system with the Vivado Hardware Manager to validate correct functionality.

C. Achieving Near Spec Performance

An overarching theme for improving either performance or productivity of FPGA designs has been build-



(a) AWS Module Layout (b) Enlarged SLR Path Crossing

Figure 8. Custom Clocking Solution for SLR Crossing

ing overlays or static shells. Overlays such as arrays of soft processors aim to raise the design abstraction, improving productivity. Static shells similar to that of the AWS F1 platform are designed to lock down high performance for a portion of the design. It is highly desirable to achieve near spec performance for these shells and overlays in order to improve the end figures of merit in the overall application. RapidWright is an excellent vehicle to build application-specific shells, especially when customers require many variations.

In the remainder of this section, we build a global data movement shell with near-spec performance leveraging RapidWright capabilities. The design, which is built on an AWS F1 instance, deploys a 128-bit LinkBlaze soft NoC [16]. The pre-implemented module layout can be seen in Figure 8a with router modules in red, forming a U-shaped network around the left side of the chip.

The maximum clock rate on a VU9P-2 in the AWS F1 instance is 775MHz, and our goal for this design was to get as close to this frequency as possible for the data movement shell. Using our pre-implemented design strategy, we identified the best area constraints and footprint sizes for the routers and network interfaces (NIs) present in the design. Ultimately, we found that combining the router with the high frequency portion of the NI produced the best modular approach for performance. We ran 88 different OOC runs per router/NI instance to identify the best implementation possible by varying various place and route parameters and providing a range of different clock uncertainty values during placement. We provided a clock constraint of 800MHz and 12 of the 88 runs successfully met the requirement. Ultimately, the best run closed timing at 811MHz when waiving the worst pulse width slack limit of 775MHz.

In our quest for near spec performance, signals crossing SLR boundaries proved to be a significant hurdle. An SLR is a die in a multi-die interposer-based device. Inter-die wires need more stringent setup and hold time requirements than intra-die fabric connections due to higher path delay variance, complicating timing closure in two ways. First, the dedicated SLR-crossing fast path between TX and RX registers experiences hold violations in most cases [17]. Second, Vivado imposes an inter-SLR compensation (ISC) penalty on all inter-die paths. We propose two strategies enabled by RapidWright to alleviate these issues.

First, RapidWright places both dedicated TX and RX registers, one on each side of the inter-die connection to minimize the delay. We resolve hold time issues by custom routing the clock in RapidWright such that each set of RX and TX registers share a common leaf clock buffer (LCB). RapidWright can then tune the LCB delay to manage the hold time issues using a time borrowing mechanism [18]. An 8% improvement over baseline resulted from this technique as reported in the second column of Table IV. An illustration of this approach for a single directional path is shown in Figure 8b.

Our second strategy involves minimizing the ISC penalty through custom clock routing in RapidWright. The ISC penalty is a 15% tax of the clocking path between a clock's root and capturing state element. As Vivado chooses a clock root to minimize global clock skew (yellow star in Figure 8a), it is often several CRs away from an SLR crossing. In contrast, RapidWright can route dedicated clock roots for each SLR crossing with one instance shown by the green star in Figure 8b, moving the relevant clock root closer by more than 4 CRs. One consequence of adding a custom clock root for each RX flop group is that time is borrowed from the following and/or preceding path and thus can become critical. To compensate, pipeline elements in the netlist are moved closer to the relevant TX and RX registers.

Ultimately, as shown in Table IV, we were able to achieve 730MHz, exceeding 94% of the device performance capabilities and a 33% boost to initial performance. This is a significant improvement to prior literature [15] which only produced overlays utilizing 27% of available device performance. In the final call to `route_design` in Vivado, congestion forces it to rip up some of the timing optimal routes for sub-optimal ones. Any efforts to preserve the timing optimal routes results in an unroutable state. This issue is similar to that of the ML design presented earlier in this section. Future work will explore this congestion issue to further push toward an implementation that meets the device spec.

Table IV
AWS LinkBlaze Design SLR Crossing Results

Vivado Baseline	LCBs	Final	Δ
549MHz	595MHz	730MHz	33%

VI. Conclusion and Future Work

We have introduced RapidWright, an open source platform that provides a new bridge to Vivado—enabling customization of emerging FPGA applications. We also proposed a new modular design methodology that implements large designs by: (1) identifying common denominator modules of the design (restructuring), (2) strategically matching pre-implemented modules with programmable fabric structures, and (3) leveraging an automated RapidWright flow that stitches modules into a final implementation. This work has demonstrated a glimpse of potential benefits such as a 50% QoR improvement or productivity improvements in compile time and debug by an order of magnitude or more.

We believe RapidWright allows the FPGA community to earnestly explore the boundaries of performance and productivity, while enjoying the credibility of commercial FPGA tools. With RapidWright enabling fine grained control of clock routing and LCB delay management, a new branch of FPGA implementation strategies can emerge—boosting performance beyond previous expectations. We believe that enabling users to capture custom crafted implementations will lead to significant improvements in circuit performance, user productivity and timing closure predictability.

We acknowledge that our methodology is more amenable to designs with high reuse and latency tolerant kernels. However, as FPGAs play a larger role in emerging and datacenter-acceleration applications, more designs exhibit these favorable attributes. In an era of push-button design flows and higher levels of abstraction, the friction of getting that first FPGA design implementation may be reduced, but delivering end design with maximum performance will require involvement from FPGA application architects.

The open source nature of RapidWright empowers application architects and power users of FPGAs to explore innovative ways to liberate the full potential of advanced silicon technology. We envision RapidWright augmented by powerful algorithmic engines—such as SAT and ILP solvers—to realize efficient, localized placement and routing solutions with the help of the FPGA community. Creating reusable, near-optimal, modular implementations will lead to shells and overlays that will unlock a class of accelerator designs rarely realized.

Author's Note: The opinions expressed by the authors are theirs alone and do not represent future Xilinx policies. To download RapidWright code, examples, tutorials and documentation, please visit www.rapidwright.io.

References

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

[2] C. Lavin, M. Padilla, J. Lamprecht, P. Lundrigan, B. Nelson, and B. Hutchings, "RapidSmith: Do-It-Yourself CAD Tools for Xilinx FPGAs," in *2011 21st International Conference on Field Programmable Logic and Applications*, Sept 2011, pp. 349–355.

[3] 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 ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '11)*, February 2011, pp. 41–44.

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

[5] S. Malhotra, T. P. Borer, D. P. Singh, and S. D. Brown, "The Quartus University Interface Program: Enabling Advanced FPGA Research," in *Proceedings. 2004 IEEE International Conference on Field- Programmable Technology*, Dec 2004, pp. 225–230.

[6] B. White and B. Nelson, "Tincr – A Custom CAD Tool Framework for Vivado," in *2014 International Conference on ReConFigurable Computing and FPGAs (ReConFig14)*, Dec 2014, pp. 1–6.

[7] T. Townsend, "Vivado Design Interface: Enabling CAD-Tool Design for Next Generation Xilinx FPGA Devices," Master's thesis, Brigham Young University, July 2017.

[8] *Vivado Properties Reference - UG912 (v2017.3)*, Xilinx, Inc., October 2017.

[9] R. LaFontaine et al., *Electronic design interchange format: EDIF. Reference manual.* Electronic Industries Association, EDIF Steering Committee, 1993, no. v. 2.

[10] *Designing IP Subsystems Using IP Integrator - UG994 (v2017.3)*, Xilinx, Inc., October 2017.

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

[12] A. Love, W. Zha, and P. Athanas, "In Pursuit of Instant Gratification for FPGA Design," in *2013 23rd International Conference on Field Programmable Logic and Applications*, Sept 2013, pp. 1–8.

[13] B. L. Hutchings and J. Keeley, "Rapid Post-Map Insertion of Embedded Logic Analyzers for Xilinx FPGAs," in *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, May 2014, pp. 72–79.

[14] E. Hung and S. J. E. Wilton, "Incremental Trace-Buffer Insertion for FPGA Debug," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 22, no. 4, pp. 850–863, April 2014.

[15] M. Yue, D. Koch, and G. Lemieux, "Rapid Overlay Builder for Xilinx FPGAs," in *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, May 2015, pp. 17–20.

[16] P. Maidee, A. Kaviani, and K. Zeng, "LinkBlaze: Efficient Global Data Movement for FPGAs," in *2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, Dec 2017.

[17] *UltraScale Architecture CLB User Guide - UG574 (v1.5)*, Xilinx, Inc., February 2017.

[18] I. Ganusov and B. Devlin, "Time-borrowing platform in the Xilinx UltraScale+ family of FPGAs and MPSoCs," in *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, Aug 2016, pp. 1–9.