IN PURSUIT OF INSTANT GRATIFICATION FOR FPGA DESIGN

Andrew Love, Wenwei Zha, Peter Athanas

Bradley Department of Electrical and Computer Engineering
Virginia Polytechnic Institute and State University
Blacksburg, VA
email: arlove@vt.edu, wenwei@vt.edu, athanas@vt.edu

ABSTRACT

This paper describes an alternative FPGA design compilation flow to reduce the back-end time required to implement a Xilinx FPGA design. Using a library of pre-compiled modules and associated meta-data, bitstream-level assembly of desired designs can occur in a fraction of the time of traditional back-end tools. Modules are bound, placed, and routed using custom bitstream assembly with the primary objective of rapid compilation while preserving performance. Since vendor tools are not needed for assembly, compilation can be performed in embedded and/or untethered environments. As a result, large device compilations can be assembled in seconds. This *turbo flow* (TFlow) enables software-like turn-around time for faster prototyping and increased productivity.

1. INTRODUCTION

Modular design is an approach that subdivides a system into smaller parts – or modules – that can be independently created and then combined and recombined into different systems. Designs for FPGAs typically consist of a hierarchy of primitive elements combining to form larger components until eventually creating a full system. The benefits of modular FPGA design become apparent when an incremental change or expansion of a design is required [1][2].

To gain the most benefit from modular design speedup, as much work as possible should be completed prior to the iterative phase of the design process (e.g., during module creation). The motivating principle behind this work is to complete as much back-end computation as possible ahead of time, even if this makes module preparation more computationally expensive. Full compilation of modules yields module-level bitstreams for later iterative assembly.

TFlow, a turbo flow for modular FPGA design, aims to reduce the design compilation time. It does so by pre-compiling modules, in many permutations, into a library. These pre-compiled modules can then be stitched together during design assembly. Use of pre-compiled components dramatically decreases design assembly time. These modules are analogous to software libraries, where pre-compiled functions are used to reduce compile time. This analogy must be extended when applied to FPGAs since additional steps of module placement/relocation, and inter-module routing is required. TFlow's pre-compiled modules can be reused in different designs, a capability not common to all modular flows. This modular reuse mimics the technique of code reuse in software development, a method proven to increase productivity [3][4].

Another productivity technique for software design is a rapid feedback loop. A user can change a design, compile it, and run it within a very short timeframe (on the order of seconds). This positively impacts the number of turns-per-day. Additionally, this capability provides a psychological change in behavior where the user may make many changes incrementally and interactively, encouraging the user to explore more design alternatives. Standard FPGA design flows can take a considerable time to complete, which reduces the feedback loop to the order of minutes to hours. In some cases, small changes to the design require full re-compilation. This reduces the number of turns-per-day, restricting the time available for prototyping. TFlow aims to reduce FPGA compilation times down to the software speed of seconds. This increase in the speed of compilation will improve the number of turns-per-day and thereby increase productivity.

TFlow gets its large productivity boost by splitting the flow into two distinct phases, as seen in Fig. 1. The first phase, referred to as the *module creation phase*, occurs when new functionality is needed. This phase is intended for an HDL programmer, or a *librarian*. This librarian designs a module that supplies this capability using a front-end tool, much like a dynamically loadable library in software development. This module design is then passed to TFlow, which shapes it and passes it into the vendor flow. This creates a bitstream and meta-data that is stored in a component library for later use.

The second phase, referred to as the *design assembly phase*, is performed by the engineer implementing the design. This designer creates high-level plans consisting of a set of modules representing a final design. TFlow will fetch the pre-created modules and assemble them using the TFlow design assembly process. The resulting bitstream can then be loaded onto an FPGA. So long as the necessary components are in the library and the library components are sufficiently parameterizable, the design loop can be

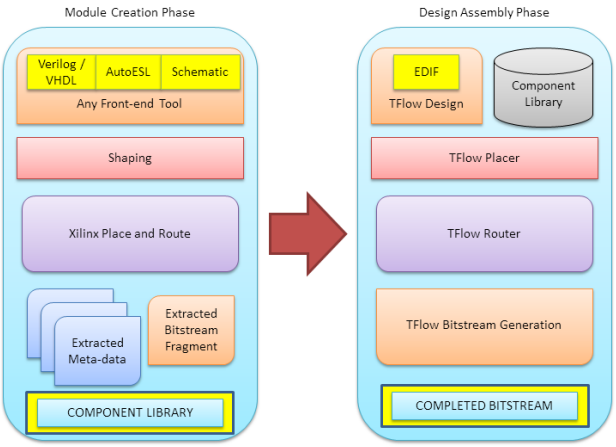

Fig. 1. TFlow Use Model.

traversed quickly for rapid prototyping. These phases will be discussed in further detail.

The starting point for the first phase of TFlow is a post-synthesis netlist in the vendor-independent standard Electronic Design Interchange Format (EDIF). Any front-end tool that ultimately produces an EDIF netlist can be used with TFlow. This includes standard HDL flows, C-to-gates flows, and graphical front-end flows. The vendor implementation flow, including the mapper, placer, router, and configuration bit generator, is run once for each module, incorporating user constraints. Key attributes of the module, such as I/O interface, anchor point, and resource usage, are extracted from the EDIF using custom tools based on the *Tools for Open Reconfigurable Computing* (TORC) [6] and stored in a meta-data file as Extensible Markup Language (XML). The vendor flow, including the appropriate bit generation tool, is used to create a module bit file. The configuration bit file and the meta-data for each module are stored in the module library.

The second phase of TFlow is design assembly. To build a design, the user only has to specify which modules to use and how to connect them. Using the same EDIF format to create a design description, the assembly flow then fetches these modules to create the design. This assembly flow goes through TFlow's placer, router, and bitfile generation phases. No time-consuming vendor tools need to be run, enhancing platform flexibility. A modified TORC router is run to make the inter-module connections. These connections are done at the bitstream level.

The compilation time saved by TFlow can be seen by comparing it to the traditional Xilinx back-end compilation flow. Starting from the initial synthesized netlist, the traditional flow has four phases. Initially, the design is mapped, which converts the post-synthesis logic gates to FPGA primitives. Then, these primitives are laid out onto the target FPGA device in the placement phase. Next, the primitives are connected together with wires in the routing phase. Finally, the bit generation phase converts the connected primitives into a final bitstream that will be used to program the FPGA. TFlow reduces the time required for each of these phases.

The mapping of logic to slices and physical components can be skipped entirely since the modules in the library have already been compiled. No additional mapping is required during assembly. The placing time is significantly reduced due to the coarser granularity of module placement versus the slice or logic placement of the standard flow. These modules have a selection of specific sizes and shapes, but are still restricted to a limited number of possible locations due to the properties of the module. This does mean that some excess area is used due to the inability to do cross-module or global optimizations. No overlap is permitted between modules. As modules were internally routed during module compilation – using vendor tools – the routing phase is reduced to inter-module connectivity. Lastly, bitstream generation time is reduced since a modified bitstream merge is performed, replacing the full vendor-provided generation process.

Four aspects of TFlow distinguish it from conventional back-end compilation:

1. It boosts the productivity of FPGA assembly by significantly reducing compilation time. This increases the number of turns-per-day possible for designers;
2. It explores the possibility of applying software engineering practices to FPGA development, in this case a library of pre-compiled components;
3. It demonstrates the power of using TORC to augment or enhance vendor-supplied compilation flows; and
4. It broadens the applicability of FPGAs to a wider selection of applications.

By speeding up the number of turns-per-day and reducing the complexity of the design process, FPGA design can begin to attract users accustomed to software design. Consequently, FPGAs can expand into fields that would otherwise choose to remain pure software, such as emerging applications like GNU Radio [7].

TFlow is not without some drawbacks. Timing for the routes that TFlow creates can be longer than that of the vendor tools, as there is no global optimization. Similarly, there may be resource optimization issues, because modules cannot share resources. Additionally, since modules cannot overlap, any unused resources in a modules footprint are unavailable to the design.

The paper is organized as follows: Section 2 will review the background of this work, including related work and a brief overview of TORC; Section 3 covers the build process for the library of components. Section 4 describes the design assembly process. Section 5 will give results, comparing three different assembly methods, and Section 6 will give the conclusions and future.

2. BACKGROUND

If you consider a progression of the densest FPGA devices over the past two decades, back-end compile time has remained nearly constant. There have been notable improvements in EDA algorithms over this period, yet these have not kept pace with device densities. In recent years, much work has been done to improve the efficiency of the front-end processes of design entry and synthesis. Xilinx has developed System Generator for DSP [8] to capture a design and convert it to Hardware Description Language (HDL). Impulse Accelerated Technologies [9] created a C-to-HDL flow for FPGAs. Instead of directly coding in HDL, high level abstractions like C or system block diagrams are successfully being used to reduce the time for design entry. Research into incremental synthesis started as early as the 1990s [10]. Commercial synthesis tools like Xilinx XST and Synopsis Symplify have long supported incremental compilation, which sharply decreases the time required to synthesize a modular FPGA design.

Back-end post-synthesis processing consumes a large portion of the full FPGA development flow. Therefore, reduction in the computation time for the back-end flow would result in the largest gains. Early work on improving this computation time was done using VPR [11], but this tool was not implemented on real devices. Incremental techniques have been exploited for single back-end steps. [12] and [13] investigate incremental techniques for the mapping stage of lookup table (LUT) based FPGAs. [14] and [15] explore incremental placing algorithms. [16] and [17] develop algorithms for incremental routing. While these techniques are effective for improving portions of the back-end flow, TFlow is focused on improving the entire flow. Another approach is to reuse pre-compiled modules, a technique that is used in [18] and [19]. Hortal and Lockwood [18] propose the idea of Bitstream Intellectual Property (BIP) cores. BIP cores are pre-compiled Intellectual Property (IP) modules that are represented as re-locatable partial bitstreams. HMFlow [19] creates pre-compiled modules that are represented as hard macros in the Xilinx Design Language (XDL), a human readable format for physical level information. As in [18], the pre-compiled modules in TFlow are represented as bitstreams. However, [18] is essentially a Xilinx Partial Reconfiguration (PR) flow and thus has limited flexibility. The modules only fit inside a few pre-defined regions. These regions are specifically for modules and are known as sandboxes. Inter-module connections must match specific bus macro interfaces with fixed routes. If a design needs a new module, the full vendor tool flow needs to be run on the whole design again, although other modules in the design may not have changed. By contrast, TFlow does not use the vendor's partial reconfiguration model; hence, it does not require fixed-location sandboxes or fixed-location bus macros. Modules can be relocated to wherever there are enough resources available and can then dynamically route the connections. New modules are compiled independently.

This contrasts with the Xilinx PR flow [20], where a module is compiled with respect to a single design framework. This framework is the static design. For a module to be used with a different design framework, it must be recompiled. TFlow can reuse modules between static designs. Additionally, Xilinx PR is intended for run-time reconfiguration, while TFlow is for fast off-line full bitstream assembly.

Another approach, *Wires-on-Demand* [21], is designed for run-time reconfiguration. It has a slot-less model with reserved routing channels between modules. TFlow is not a run-time reconfiguration tool, and it is also a more general solution to modular design, as it does not require reserved routing channels.

QFlow [22] uses a similar method to TFlow, but leaves room for improvement in its speed optimization. Whereas TFlow generates modules for the library at the bitstream level, QFlow stops after the map stage. These modules must then be routed at run-time, slowing down design assembly. By using mapped modules, QFlow gains flexibility at the cost of speed.

Both HMFlow [19] and TFlow make use of XDL. HMFlow stores all of the module information in XDL, including logical instances, their placements, and their routing. TFlow, however, mainly uses XDL to as an input mechanism for its meta-data, extracting physical and net information. More importantly, to create a full design, HMFlow must convert the final design XDL into a Netlist Circuit Description (NCD), a Xilinx physical description file, before creating a bitstream for use on a device. This XDL-to-NCD conversion and subsequent bitstream generation takes considerable time and scales to the size of the design. Tests show that this overhead exceeds the total TFlow runtime, acting as a performance bottleneck. Instead, TFlow, like QFlow, uses a different approach that does not require this costly XDL-to-NCD conversion process and thereby speeds up bitstream creation.

2.1. TORC

TFlow relies heavily on TORC [6], an open-source C++ infrastructure and tool-set for reconfigurable computing. The TORC infrastructure is able to read, write, and manipulate EDIF, Berkeley Logic Interchange Format (BLIF), and XDL netlists, as well as Xilinx bitstream frames. The TORC tools include placing and routing capabilities for full or partial designs, along with additional capabilities to facilitate design manipulation and analysis.

Many of the TORC APIs and tools are used by TFlow. The EDIF importer extracts a module's logical level information for use in creating TFlow meta-data. The XDL importer extracts physical information from the module's XDL, including anchor point, shape, and routing information. TORC also contains a device database (DDB) that can track wire and logic resource usage information for

a wide range of target devices. Importantly, TORC also includes a router that can treat previously used wires as constraints to avoid contention. The bitstream parser can map from the frame indices of a bitstream file to the frame addresses on a device.

2.2. Module Relocation

Module relocation is an important component of TFlow. FPGAs consist of different types of tiles, such as Configurable Logic Blocks (CLBs), Block RAM (BRAM), and Digital Signal Processors (DSPs). These tiles are arranged in a regular pattern throughout the device. TFlow leverages this regular structure for module bitstream relocation.

A Xilinx FPGA is configured by loading a bitstream file. This file is organized into frames, the smallest addressable segment of the Virtex-5 configuration memory space [23]. A frame address maps to a tile on the FPGA, and is represented as 32 bits. Multiple frames may point to different portions of the same tile.

By manipulating the frame address, the frame data can be moved around the device. Knowledge of the contents of a frame is unnecessary for this bitstream level relocation.

Other Xilinx FPGA families, such as the Virtex-4, have frame addresses that work in a similar way. Several research teams have demonstrated methods for module bitstream relocation. [24] uses frame relocation as part of its fault tolerance tool for the Virtex-II Pro. Becker [25] discusses a way to do more flexible bitstream relocation on the Virtex-4. Becker's work allows module relocation onto regions with different resources at the expense of underutilizing the region.

Module relocation is used in the placement phase of TFlow, and is discussed in detail in Section 4.2.

3. MODULE CREATION PHASE

One of the key advantages of TFlow is the library of pre-compiled components for later assembly. These components come as either standard pre-compiled modules or top-level pre-compiled static designs.

3.1. Module Creation

The entry point for TFlow module creation is a post-synthesis EDIF. EDIF files are an open standard that can be automatically created by most front-end tools. EDIF contains a logical level representation of a module, including connectivity, logic, ports, and other information. TFlow compiles the module through an enhanced version of the Xilinx Partition Flow [26]. Once this flow is complete, the module and its associated meta-data are added to the library.

3.2. Module Shaping

Before compiling a module, an appropriate shape must be selected. The shape and resource utilization of a module will decide how it can be integrated into a final design. To choose a shape, an estimation of the number and type of resources is required. TFlow uses PlanAhead for resource estimation. TFlow's custom shaping tool then creates a minimum footprint for the module that meets both resource and TFlow-specific requirements. TFlow has additional shaping rules that improve area utilization during design assembly.

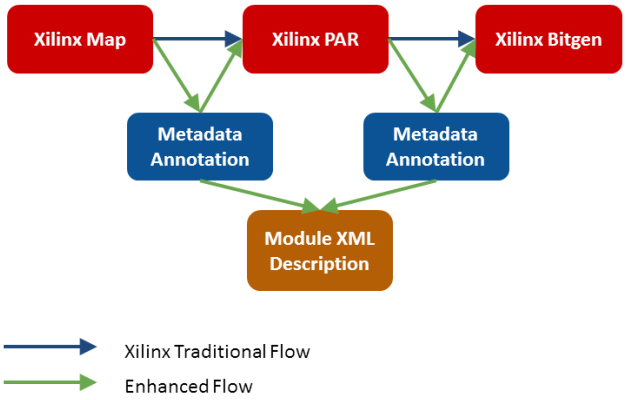

Fig. 2. Meta-data Annotation Process.

The resulting region is given in the form of a Xilinx User Constraints File (UCF). This file will be used as a constraint for the Xilinx Partition Flow.

3.3. Module Compilation

TFlow uses an enhanced version of Xilinx Partitions for compilation. Unlike the standard Xilinx tools, Xilinx Partitions will enforce routing constraints. Module routing will therefore remain inside the shaped region. This results in a reduced footprint for each module, improving module packing.

Additionally, Xilinx Partitions automatically performs port consolidation. Port consolidation takes a large set of inputs to a module and reduces them to a single port using bus macros. This decreases the fan-out required for the final assembly stage; this reduction in routing complexity results in increased speed when routing the design, at the cost of a slightly longer combinatorial path.

In addition to the module bitstream, TFlow will also generate meta-data describing both the physical and logical properties of the module. The initial logical-level data extraction occurs from the module EDIF. Physical-level information is obtained from the module XDL. The flow for creating meta-data for a module can be seen in Fig. 2.

Modules created for TFlow have a few additional requirements. The foremost requirement is that modules register all input and output signals. This removes inter-module timing issues that may otherwise occur. One property of modular design is that timing closure is most often an issue within the module, and does not span across modules. The vendor place-and-route tools are used for this intra-module routing to ensure timing is met.

3.4. Static Creation

The other library component is a top-level static design. This design contains the logic that will not change and is mainly used as an interface to the modules. For example, I/O ports, memory controllers, or Ethernet interfaces would be good candidates for inclusion in the static design.

A static design requires additional bus macros – to interface with the modules – and a module sandbox at both logical and physical levels. The physical-level sandbox is created during the static compilation process. This physical sandbox will be completely void of any logic or routing; thus providing a clean region for module placement.

Compiling the static is done using the normal Xilinx tool-chain, starting with an EDIF file and resulting in a bit-stream. As with module creation, TFlow will create meta-data representing the static design.

3.5. XML Meta-Data

Since the modules are stored in the library as bitstreams, meta-data describing these modules must contain all of the necessary information for implementation. This meta-data describes the module at both the logical level and the physical level. Neither XDL nor EDIF contain a complete description of the module, but by combining information from both, a full picture can be obtained. Some of the information contained within the meta-data includes port information, clock names, and utilization boundaries. Port information consists of the physical and logical names of the ports for later translation of high-level connectivity into physical-level routes. The position and utilization boundaries are required for module placement during design assembly. By performing this data extraction prior to design assembly, design compilation time can be reduced.

4. DESIGN ASSEMBLY PHASE

The design assembly process can proceed quickly due to the pre-built library of components. This section describes the major steps in the iterative design creation phase.

4.1. Design Entry

Any front-end tool that generates an EDIF file can be used to create a design for TFlow assembly. One such tool is GReasy [7]. GReasy is a TFlow enhanced version of the GNU Radio [27] environment, an open-source environment for software-defined radios. Many radio applications could be improved by using FPGAs, but the target audience is software designers. By having a librarian with FPGA knowledge create the radio components, these software designers can use TFlow to enhance their designs with FPGAs without programming HDL. To keep the librarian/designer divide, GReasy automatically creates the design entry EDIF from its graphical user interface.

The design assembly EDIF file specifies the modules and their connectivity. The connectivity information details how the modules connect to one another and to the static design.

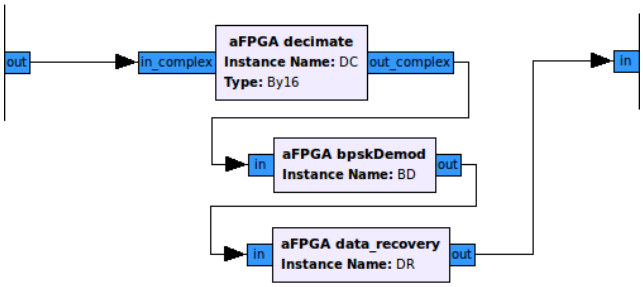

Fig. 3. Design Connectivity Example.

Fig. 3 shows the graphical user interface for a GReasy radio design. The connectivity information is shown as multi-bit wires. The static interfaces are seen at the edges of the design. This design implements a BPSK radio from library components. The data path goes from the static, through three components, and then back into the static interface. This high-level description of a design will use TFlow to implement itself on an FPGA, without the user needing to know FPGA design or module implementation details.

To process this EDIF, TFlow fetches the meta-data for each of the components from TFlow's pre-built library. This information includes the ports and resource requirements. The static design information, including available space for placement, is also fetched.

4.2. Module Placement

Once the modules have been fetched from the library, they must be assigned a location inside the static's sandbox region. The placement tool [22] uses TORC as a framework, as TORC's databases contain information about the device resources and structure of the FPGA. The placer then proceeds to find valid module locations. Modules have placement restrictions because each primitive, such as BRAMs and DSP48s, must be properly matched. To do so, the module's placement must match in both resources and routing. Additionally, due to the frame requirements for bitstream relocation discussed in Section 2.2, the module must maintain the same alignment within a clock region as when it was originally built. Therefore, the module can only move vertically in jumps equal to the size of a clock region on the FPGA. There is no such clock restriction on horizontal relocation. These restrictions reduce the granularity of the placement space. Precompiling several versions of a module, each with a different shape, provides more flexibility during module placement. Modules may not collide with one another to share resources; this may increase the size of the final design. A more detailed analysis of the placer can be found in [22].

After all modules have been placed, optimizations are made to minimize connection length. The completed module placement is then added to the meta-data, which is then used during final assembly to (a) relocate the

component in the bitstream, and (b) identify the exact position of the module's terminals for subsequent routing.

This placement step can thus be completed in seconds due to the large granularity of the placement problem.

4.3. Clock

The clock information is extracted from the static meta-data and incorporated into the design. A clock router will create the necessary connections as part of the inter-module routing.

4.4. Inter-module Routing

Once the modules are placed, the next step is to route the desired inter-module connectivity. TORC's routing capability [6] was expanded into a router for TFlow. The terminals of the pre-compiled modules and their inter-module connectivity form a routing task list. TFlow's custom router then generates a list of the PIPs necessary to route the design. This custom router is designed primarily for execution speed, routability, and lastly, timing performance. As mentioned in Section 3.3, the I/O signals of the modules are registered, so the inter-module timing constraints are lessened. The PIP listing is then passed to the next phase of TFlow for transformation into bits.

4.5. Bitstream Stitching

The final step creates a bitstream that implements this design. The static bitstream is fetched from the library as a starting point. As mentioned in Section 3.4, this static bitstream has clean regions that have no logic or routing. These are the sandbox regions where the modules are to be placed. The meta-data specifies the module bitstream frames for relocation into the static bitstream. This overwrites the region, which is why the sandbox region must be empty; otherwise, existing logic will be erased. The contents of the frames are not changed for relocation. Lastly, the connectivity PIPs for routing are translated into bits. Writing these bits readies the bitstream for transit onto the physical device. See [28] for more details about bitstream generation. When dealing with assembly at the bitstream level, no additional processing, such as XDL-to-NCD conversion or bit generation, is necessary, in contrast to flows like QFlow or HMFlow [19].

5. RESULTS

A full version of TFlow has been built for the Xilinx Virtex-5. The speed advantage and flexibility of TFlow is shown in the following results.

Results were measured by starting from identical netlists and running each tool until bitstreams were generated. Three different tool flows were used. The first tool is the traditional Xilinx ISE tool flow, from netlist to bitstream. This is the flow that most designers would normally use, and can be seen as a control. The second tool is QFlow, which uses a library of unrouted hard macros. These modules are then placed and routed at design time. Routing and bitstream generation is done using the Xilinx vendor tools. The final tool is TFlow.

Table 1. Assembly Time for BPSK Radio (seconds).

	Map	Route	Bit gen	Total
ISE	109	47	34	284
QFlow	43	49	42	134
TFlow	11	6	–	17

The first tests were run on a GReasy [7] design targeting the XC5VLX110T board. GReasy automatically generates the high-level EDIF required for design entry from a GUI interface. QFlow can also use this EDIF as a design entry point. For the ISE design, design entry begins with a netlist in the form of a Xilinx Native Generic Circuit (NGC). Table 1 splits up the assembly time into its component portions, showing the effect of each on the total time.

This test consists of a full BPSK Radio with its associated modules for placement in a static design, with the results seen in Table 1. The first step, map, has results which reflect the differences in granularity between the tool flows. Where ISE has no reduction in granularity, QFlow places pre-mapped modules into the sandbox based on resource constraints. TFlow's modules are both pre-mapped and pre-routed, so they have a reduced number of possible valid locations for placement. Bitstream frame alignment issues reduce the granularity further, resulting in considerable speedup. For the routing stage, both ISE and QFlow use the Xilinx Place and Route tool, yielding similar results. TFlow only routes inter-module connectivity during design time due to its pre-routed modules. Lastly, bitstream generation for both ISE and QFlow use the Xilinx Bitgen tool, whereas TFlow has integrated bitstream generation in its routing stage. QFlow has a speed advantage over the Xilinx Flow, while TFlow has a clear overall speed advantage.

The following tests show total runtime for each flow on four different test cases. The library is already populated with the modules and the static components.

The first three test cases target the Xilinx Virtex-5 XC5VLX110T FPGA board. The initial design is for a video edge filter. As can be seen in Table 2, TFlow performs approximately eight times faster than the other flows. The second design is created by swapping out the edge filter for a video Gaussian filter. Assembly of the Gaussian filter also has an 8.8-9x speedup over the other methods. The time required to run TFlow is the total time from having an edge filter design to having a Gaussian filter design, because both of these designs share a static. The third design has a different static and uses modules for a ZigBee Radio. This static is more complex resulting in a more pronounced difference between QFlow and ISE. However, TFlow maintains its lead in all three cases. This lead is due to the significant pre-processing of TFlow's components, reducing the computation necessary for assembly.

Table 2. Design Assembly Times.

	Time (s)			TFlow Speedup	
	ISE	QFlow	TFlow	Over ISE	Over Qflow
Edge Filter	184.6	170.8	21.6	8.5x	7.9x
Gaussian Filter	159.8	156.7	17.8	9.0x	8.8x
ZigBee Radio	236.2	157.7	23.8	9.9x	6.6x
Vector Add	3891.7	805.1	98.7	39.4x	8.2x

Table 3. Total Design Area Utilization.

	SLICE		DSP		BRAM	
	ISE	TFlow	ISE	TFlow	ISE	TFlow
Edge Filter	2865	2838	14	14	0	0
Gaussian Filter	2062	2068	18	18	0	0
ZigBee Radio	1648	2028	10	10	8	9
Vector Add	22444	11443	0	0	53	60

The fourth test case was run targeting a Xilinx Virtex-5 XC5VLX330 board. This board is considerably larger and the static design is for the more complicated Convey environment [5]. A vector-add module was used for this test. With this more complex design, the differences between the flows are emphasized. While QFlow has some significant gains over ISE, TFlow completes assembly in ninety-nine seconds. This results in a speedup of almost forty times that of ISE, as shown in Table 2. The static and the routing for this design are very complicated, but TFlow's use of pre-compiled modules allows for quick and flexible modular design.

The area overhead for implementing TFlow relative to ISE can be seen in Table 3. QFlow has the same area utilization as TFlow and thus is not included. The results are for the total area of the final design, including both the static and the modules. Because TFlow's module and static shaping attempt to compress logic density, very little area overhead is required. This increased logic density can be best seen in the Convey vector-add design, where there is a large reduction in slice utilization. As ISE will perform global optimizations, it will expand into unused areas for timing or compiler performance reasons. From these results, it can be seen that the area overhead for using TFlow's modular design is slight, and the shaping process can compress designs efficiently.

6. CONCLUSION

TFlow can reduce compile times dramatically, allowing for turn-around times comparable to software programming. This allows for modern programming techniques, such as spiral or waterfall models, to be used in the FPGA environment. Quicker prototyping on a platform designed for rapid development can lead to additional opportunities for FPGA adoption.

Some limitations to TFlow that could be addressed in future work include static design selection and improved version support. Partitioning a design into the static component and the modules remains a difficult problem. This partition determines what logic can be changed easily – the modules – and what logic changes will require full re-compilation – the static. Additionally, timing-sensitive connections must not be split between components. This results in a trade-off of speed and timing for better granularity, since larger modules have better internal timing, but a reduced placement space. Thus, component partitioning can play an important part in creating good designs for TFlow.

Additionally, the current iteration of this tool flow only supports Xilinx Virtex-4 and Virtex-5 devices. Future work targeting the Xilinx Zynq-7 is promising, and would help adoption of this flow as a productivity tool.

One topic not addressed in this paper is the possibility of creating modules that allow a degree of parameterization. This capability would greatly reduce the number of module permutations required. This is a topic that is currently under investigation.

Integration of this tool with GNU Radio [7] has shown the power of this technique and its ease of adoption in an application environment. Modules can be created by FPGA designers and added to the library, where users not proficient with HDL design can then profit from the myriad advantages of FPGAs. This abstraction of FPGA design into a higher level process allows for non-FPGA developers to gain the benefits of FPGAs without the long compile times and learning curve normally necessary. TFlow is leading the way by allowing this kind of abstraction at the assembly level.

7. REFERENCES

[1] E. Hung, S. Wilton, "Limitations of incremental signal-tracing for FPGA debug," *Field Programmable Logic and Applications (FPL 2012)*, pp.49-56, Aug. 2012

[2] D. Grant and G. Lemieux, "A spatial computing architecture for implementing computational circuits," *Microsystems and Nanoelectronics Research Conference, 2008*, pp.41-44. 2008.

[3] H. Mili, F. Mili, and A. Mili, "Reusing software: issues and research directions," *IEEE Transactions of Software Engineering*, vol 21, pp. 528-562, 1995.

[4] W. Frakes and K. Kang, "Software Reuse Research: Status and Future", IEEE Transactions on Software Engineering, vol. 31, no. 7, 2005, pp. 529-536.

[5] Convey-HC1 processor datasheet [Online]. Available: http://www.conveycomputer.com/Resources/HC-1%20Data%20Sheet.pdf

[6] 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.

[7] R. Stroop, "Enhancing GNU Radio for Run-Time Assembly of FPGA-Based Accelerators," Master's Thesis, Virginia Tech, 2012. [Online]. Available: http://scholar.lib.vt.edu/theses/available/etd-08082012-154538/

[8] Xilinx, “UG640: System Generator for DSP,” 2012. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_4/sysgen_user.pdf

[9] Impulse Accelerated Technologies, http://www.impulseaccelerated.com/

[10] M. Lehky and S. Bilik, “Reducing FPGA design modification time,” *Proceedings of VHDL International Users’ Forum*, pp. 143-149, 1997.

[11] 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*, 1997, pp. 213-222.

[12] J. Cong and H Huang, “Depth Optimal incremental mapping for field programmable gate arrays,” in *Proc. ACM/IEEE Des. Autom. Conf.*, Los Angeles, CA, Jun. 2000, pp. 290–293.

[13] M. Teslenko and E. Dubrova, “Hermes: LUT FGPA technology mapping algorithm for area minimization with optimum depth,” *ICCAD-2004*, pp. 748-751, 2004.

[14] D.P. Singh and S.D. Brown, “Incremental placement for layout-driven optimizations on FPGAs,” *ICCAD-2002*, pp. 752-759, 2002.

[15] D. Leong and G.G.F. Lemieux, “Replace: An incremental placement algorithm for field programmable gate arrays,” *FPL 2009*, pp. 154-161, 2009.

[16] J.M. Emmert and D. Bhatia, “Incremental routing in FPGAs,” *ASIC 1998*, pp. 217-221, 1998.

[17] E. Keller, “Jroute: A Run-Time Routing API for FPGA Hardware,” *7th Reconfigurable Architectures Workshop*, pp 874-881, 2000.

[18] E. Hortal and J. Lockwood, “Automated Method to Generate Bitstream Intellectual Property Cores for Virtex FPGAs,” *In Proc. Field Programmable Logic.2004*, 2004.

[19] C. Lavin, M. Padilla, J. Lamprecht, P. Lundrigan, B. Nelson, B. Hutchings, “HMFlow: Accelerated FPGA Compilation with Hard Macros for Rapid Prototyping,” *FCCM 11*, pp. 117-124, May 2011.

[20] Xilinx, “UG702: Partial Reconfiguration User Guide,” 2010. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx12_1/ug702.pdf

[21] P. Athanas, J. Bowen, T. Dunham, C. Patterson, J. Rice, M. Shelburne, J. Suris, M. Bucciero, J. Graf, “Wires on Demand: Run-Time Communication Synthesis for Reconfigurable Computing,” in *International Conference on Field Programmable Logic and Applications*, pp. 513–516, 2007.

[22] T. Frangieh, P. Athanas, “ A design assembly framework for FPGA back-end acceleration,” in *Reconfigurable Computing and FPGAs (ReConFig 2012)*, pp. 1-6, 2012.

[23] Xilinx, “UG191: Virtex-5 FPGA Configuration User Guide,” 2012. [Online]. Available: http://www.xilinx.com/support/documentation/user_guides/ug191.pdf

[24] D.P. Montminy, R.O. Baldwin, P.D. Williams, B.E. Mullins, "Using Relocatable Bitstreams for Fault Tolerance," *Adaptive Hardware and Systems, 2007. AHS 2007. Second NASA/ESA Conference on*, pp.701-708, 5-8 Aug. 2007.

[25] T. Becker, W. Luk, P.Y.K. Cheung, "Enhancing Relocatability of Partial Bitstreams for Run-Time Reconfiguration," *Field-Programmable Custom Computing Machines, 2007. FCCM 2007. 15th Annual IEEE Symposium on*, pp.35-44, 23-25 April 2007.

[26] C. Zeh, "Incremental Design Reuse with Partitions," Xilinx XAPP918 (v1.0), June 7, 2007. [Online]. Available: http://www.xilinx.com/support/documentation/application_notes/xapp918.pdf

[27] GNU Radio Website. March 2013. [Online]. Available: http://www.gnuradio.org

[28] R. Soni, N. Steiner, M. French, “Open-Source Bitstream Generation”, *FCCM 13*, pp. TBD, April 2013.