A Design Assembly Framework for FPGA Back-End Acceleration

Tannous Frangieh and Peter Athanas
NSF Center for High-Performance Recongurable Computing (CHREC)
Virginia Tech Configurable Computing Lab
Department of Electrical and Computer Engineering
Blacksburg, Virginia (USA)
Email: {tannous,athanas}@vt.edu

***Abstract*—Long wait times constitute a bottleneck limiting the number of compilation runs performed in a day, thus risking to restrict FPGA adaptation in modern computing platforms. This work presents an FPGA development paradigm as a means to increase FPGA productivity. The practical tasks of logic partitioning, placement and routing are examined and a resulting assembly framework, qFlow, is implemented. Experiments show up to 6x speed-ups using the proposed paradigm compared to the vendor tool flows.**

***Keywords*-FPGA; Modular-Assembly; Back-End Acceleration; Productivity**

I. Introduction

Initially used to implement glue logic, FPGAs today are ubiquitous in automotive, data centers, high performance computers, medical, networking, security and asic prototyping, to name but a few [21], [4]. The speedup and modest power consumption of FPGA-based systems however, are often associated with long wait times before a device configuration is generated. Compared to contemporary alternatives such as microprocessors and GPUs, FPGA development tools can take hours if not days to generate a configuration, greatly hampering FPGA productivity and risking to limit device adaptation in modern computing platforms.

An FPGA design process consists of two complementary phases: a front-end phase and a back-end phase. A designer models and simulates their problems before hardware implementation, using tools built to describe hardware systems, i.e., the front-end tools. The design model is then synthesized into a structural netlist as the last stage of the front-end process. Today, there exist two classes of front-end tools: graphical (such as LabVIEW and Azido [6]) and textual (such as VHDL and Catapult C [12]). Thereafter the back-end tools consume the resulting netlist and turn it into a device configuration. Vendor specific, the back-end tools solve difficult problems associated with transforming the netlist to a FPGA configuration. The algorithms for solving these problems typically do not scale well as the netlist size grows, making this a time intensive process. MAP and Place and Route (PAR) [20] are examples of Xilinx back-end tools, which typically produce high-quality results. Due to their long run-times, back-end tools are the major contributor to the FPGA productivity problem. Table I highlights back-end compilation times for some FPGA designs.

TABLE I
FPGA COMPILATION TIMES FOR SAMPLE DESIGNS

Design	Platform	Impl. Time (s)
Edge Detection	Xilinx ML 509	270
Gaussian Blur	Xilinx ML 509	350
ZigBee	Xilinx ML 509	441
1 Vector Addition	Convey HC-1	4084
3 Vector Additions	Convey HC-1	4530
Molecular Dynamics (GEM)	Convey HC-1	50400

Irrespective of the long run-times, vendor tools excel in the fidelity of their results, while maximizing device utilization and clock speed. In the process of optimizing designs, the tools flatten carefully crafted hierarchical designs, and re-implement verified and implemented logic for every design iteration, incurring longer compilation times. For many FPGA designers, vendor tools serve the purpose of generating highly optimized designs that can fit on smaller, thus cheaper, devices. Practically speaking, FPGA back-end compilation constitutes a bottleneck limiting the number of iterations performed in a day. The fact that back-end tools take hours or days to generate a configuration, shifts the designer focus from the computation itself, to irrelevant recurring long waiting times, and jeopardizes their progress on solving problems.

This work presents a way of viewing FPGA development, in which FPGAs are treated as flexible hardware capable of quick adaptations to computational needs, not only as reconfigurable devices operating optimized high-speed designs. In this paradigm, a design is split into two complimentary sets based on logic variance: a mostly invariant logic set and an evolving logic set. Compared to a holistic design implementation by current FPGA development methodologies, the work implements each of the two sets separately. As the name indicates, the *invariant set* is less likely to change over the design process, thus less frequently re-implemented. The *evolving set* is oftentimes re-implemented during design iterations. Through separation of concerns and re-use, the approach speeds up the implementation process and put the designer's focus on the creative part (algorithms and computations) of the design process.

The rest of the paper is organized as follows. Section II summarizes related work. Section III describes the proposed assembly paradigm. In Section IV, we implement a framework based on the proposed paradigm. Results are discussed in Section V. Finally, in Section VI we conclude and give directions into future work.

II. Related Work

Productivity issues related to FPGA back-end compilation have received quite a bit of attention from both industry and academia in recent years. This section summarizes some of the most relevant activities. Xilinx currently offers three designs flows, all of which belong to Xilinx Hierarchical Design (HD) flows: Team Design flow, Design Preservation flow and Partial Reconfiguration (PR) flow. Using Team Design flow [14], a team of designers can concurrently work on a design. A design is initially divided into modules that are floorplanned, then each team member implements their portion of the design concurrently and independently of other modules. An assembly step stitches the different parts together into one implemented design. Design Preservation flow preserves compilation results and reuses them in future compilation runs, thus speeding up the compilation process. The process is enabled through the Xilinx Partitions flow, supporting three level of preservation: synthesis, placement and routing logic preservation. Xilinx PR flow [1] enables the partial modification of an operating PR design by loading partial configuration(s), while other parts of the design are still functioning. The modification is performed on a dynamic region of the design, while other static parts of the design remain intact. Altera has its own implementation of the different HD flows, including Altera's Team-Based Design, Incremental Compilation with Design Partitions and PR. For more information, the reader is referred to Quartus II Handbook Volume 1: Design and Synthesis [2].

JBits [13] is an API into Xilinx FPGAs configuration bitstreams. Written in Java, JBits provides the capability to design and modify complete circuits for Xilinx FPGAs in a software development environment, yielding quick edit/compile cycles. JBits speeds up the configuration generation by completely skipping the back-end tools. Wires-on-demand (WoD) [5] implements an efficient run-time reconfiguration framework for Xilinx FPGAs. The framework allocates a sandbox on the device where precompiled modules fetched from a library can be placed and routed at run-time, with a software layer abstracting the low level reconfiguration details away from the designer. WoD caches configuration for all modules skipping the back-end tools during assembly. In [7] the authors presents an FPGA CAD tool, *Block Place and Route*, that pre-computes internal placement and routing of reused cores to speed-up the implementation process. PATIS [9] is an automatic floorplanner that leverages partial reconfiguration to improve implementation and debug turnaround. The tool incremental approach and localization of design updates to a corresponding partial reconfiguration region itself lead to a faster design convergence and configuration generation. HMFlow [15] is another FPGA design flow based on hard macros that builds on RapidSmith [16]. The flow implements a custom mapper, placer, and router and leverages hard macros and caching to speed up the FPGA compilation process. Some consider PR flows have a productivity aspects to them. OpenPR is an open-source, slotbased, partial-reconfiguration toolkit for Xilinx FPGAs [18]. The toolkit is composed of several utilities written in C++ tied together via scripts and Makefiles, and packaged in a form that makes it ready to use for traditional slot-based partial reconfiguration. Finally, the modular-based assembly from [10] provides an environment for FPGA application developers, that pre-compiles computational units in an application then stitches the precomputed modules into a physical netlist.

Two major weaknesses govern most of the flows list above: architecture dependence and stiffness. An architecture dependent implementation flow is oftentimes optimized for that architecture yielding considerable amount of speedups for that architecture at the expense of porting to different architectures. A stiff implementation flow finds it hard to adapt to design changes, such as logic relocation on the device.

III. Theory: A Proposed Assembly Paradigm

We propose a design assembly paradigm for FPGA development that exploits design variance and logical hierarchy. The proposed paradigm partitions a design into two classes, an invariant set and an evolving set, incorporating different implementation techniques and frequency for the two classes. Following the proposed partitioning, the invariant set is less likely to change, thus less frequently re-implemented, whereas the evolving set is oftentimes updated and re-implemented. The proposed flow addresses the two major weaknesses of existing flow through device architecture independence and flexible logic implementation. Contrary to current tool flows that follow a holistic approach when implementing a design, the proposed technique divides the implementation process into four phases:

1) Design Partitioning
2) Invariant Set Implementation
3) Evolving Set Implementation
4) Design Assembly

Figure 1 presents a high-level view of the proposed assembly paradigm. Details about each of the different phases of the paradigm follow.

A. Design Partitioning

The first step in the implementation process is the design partitioning phase. During this phase, a design is partitioned into invariant and evolving logic. Logic can move back and forth between these partitions over the course of development, yet the general concept is to cluster all the logic that is currently being refined into the evolving logic partition. Figure 2 shows an example partitioned design.

B. Invariant Set Implementation

The invariant set implementation process allocates a *sandbox*, a region on a device that excludes logic and routing

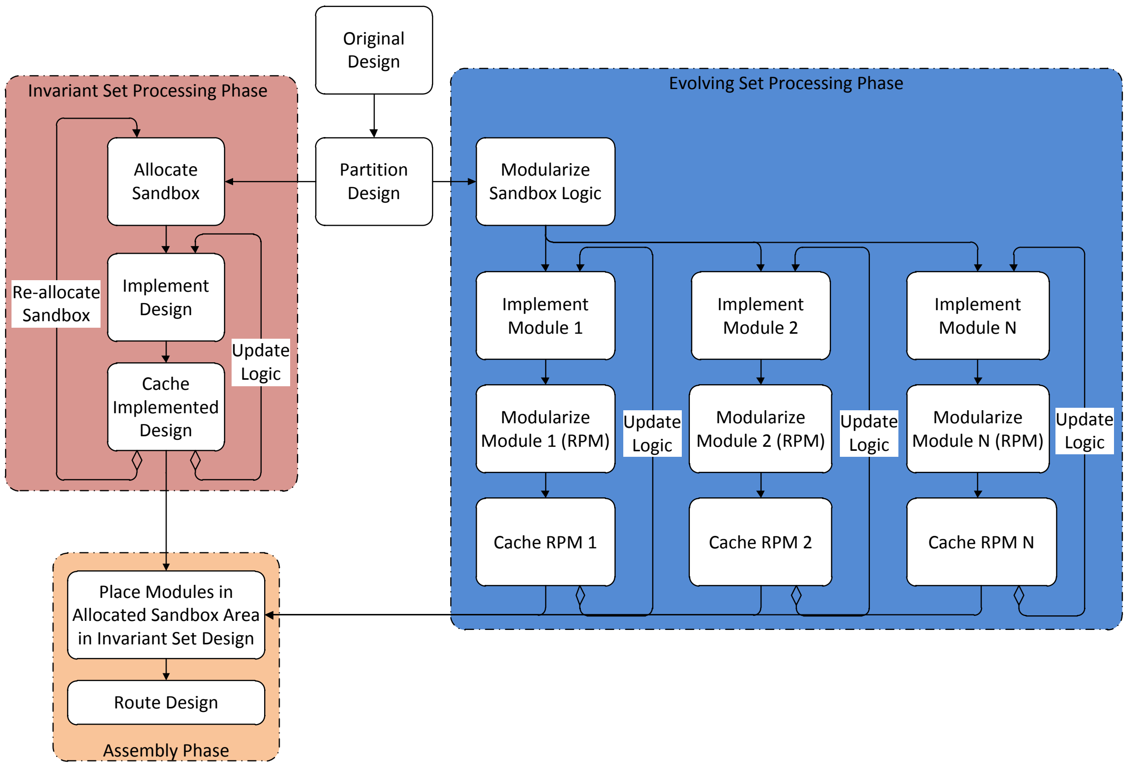

Fig. 1. A high-level view of the proposed assembly paradigm

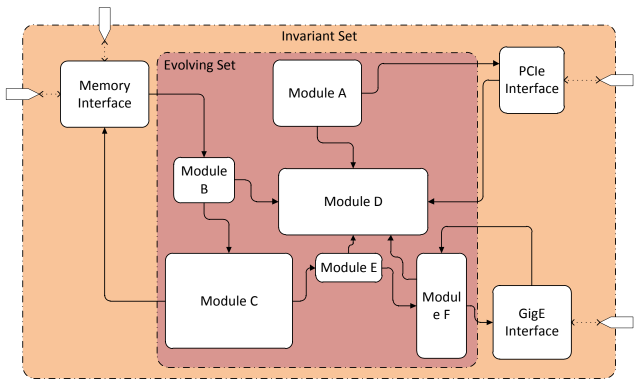

Fig. 2. Partitioned Design

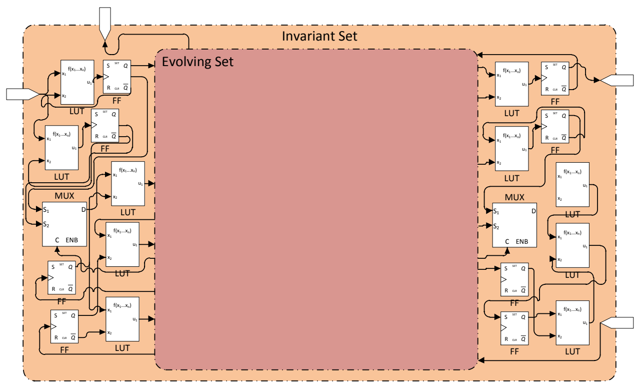

Fig. 3. Invariant logic physical netlist

resource utilization, that will eventually host the evolving logic. To yield a successful design assembly, the allocated sandbox area needs to satisfy the minimum resource requirements for evolving logic. The vendor tools are then called to map, place and route the invariant logic. The outcome of this phase is a placed-and-routed design with dangling wires to the evolving set. Figure 3 illustrates an implemented invariant logic example with an allocated sandbox. The implemented invariant set physical netlist is then cached and retrieved during the design assembly phase, but never re-implemented until its corresponding logic changes, sparing the designer considerable amount of time. It is assumed that re-implementing the invariant set is much less frequent than re-implementing the evolving set.

C. *Evolving Set Implementation*

During the evolving set implementation process, every module of the evolving logic set, assumed to be hierarchical, is separately floorplanned, mapped, placed and routed. The implementation result is represented as a relatively placed macro (RPM), a representation that provides structure to

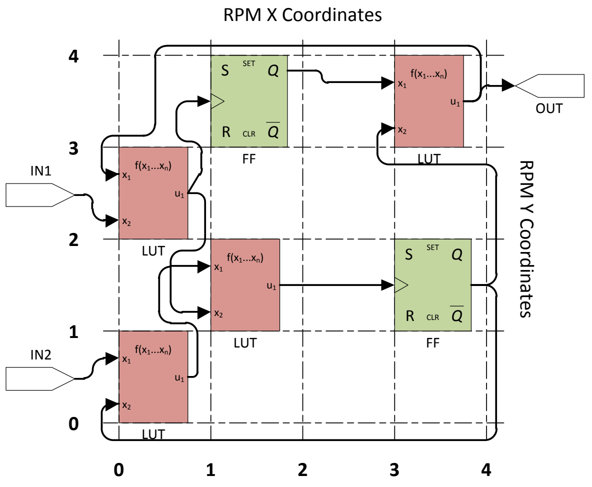

Fig. 4. Evolving module implementation (RPM)

the design elements without the need to specify absolute placement location on the device [11]. Although the technique is originally meant to help implementation tools meet timing [11], the main purpose of using RPMs is their ability to easily relocate modules on the device. All module RPMs are then cached into a library for later retrieval, when needed during the design assembly phase, saving re-implementation time of their corresponding modules. With a hierarchical evolving set, the effect of a change to a module is local to the module itself and only the updated module is re-implemented. Figure 4 illustrates an example evolving module implementation in RPM representation.

D. Design Assembly

The final phase of the implementation process is the design assembly. During this phase, the implemented invariant set produced from Section III-B is assembled with the implemented evolving set produced from Section III-C, by placing the corresponding evolving logic RPMs in the sandbox region reserved during the invariant set implementation phase. A routing stage finalizes the assembly and produces the final design. Details about placement and routing during the assembly phase follow.

1) Placer: Bounded by the sandbox resources and its shape, the evolving set RPMs required resources and their shapes, a placer places the evolving logic set by computing an absolute value for each module's RPM. The placement process runs until all modules are placed or until the placer fails to find a feasible solution. Several factors can cause the placer to fail such as insufficient sandbox resources or improper RPM floorplan. Different placement algorithms can also affect the feasibility, the quality as well as the convergence speed of the solution [17]. Figure 5 shows the first stage of the assembly with two evolving RPMs placed in the allocated sandbox.

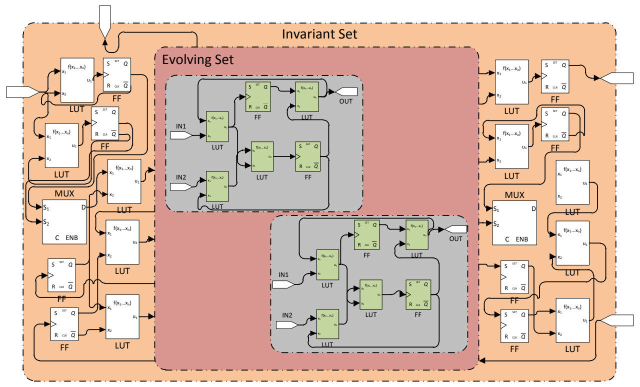

Fig. 5. Two evolving logic modules placed in the invariant logic implementation allocated sandbox

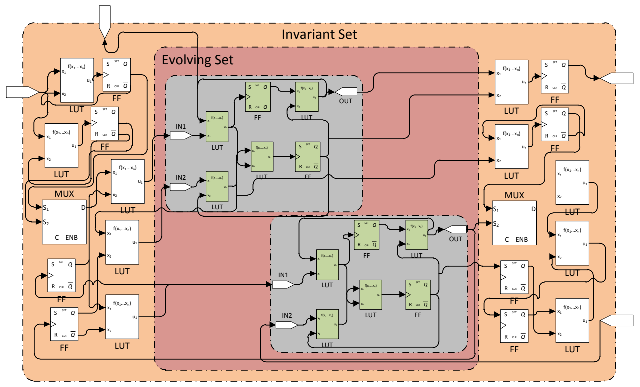

Fig. 6. A fully placed and routed design

2) Router: After a successfully placed design, a router traverses the evolving set connectivity list and routes all intermodule nets. The invariant set connectivity is already handled in Section III-B. The routing process runs until all nets are routed or until the router fails to find a feasible solution. The router fails mainly due to insufficient routing resources in congested areas on the device. Different routing algorithms can also affect the feasibility, the quality as well as the convergence speed of the solution [17]. The routing phase ends by running a timing analysis step against all design timing constraints to verify that the assembled design meets timing. The resulting design is passed to the vendor tools to generate a configuration for the target device. Figure 6 shows the resulting assembled design.

IV. Implementation: Quick Flow (qFlow)

In this section we present *qFlow*, a back-end productivity framework that implements the concepts presented in Section III. Implemented in C++, qFlow is an FPGA back-end productivity framework that builds on tools for open reconfigurable computing (TORC) [19], an open-source infrastructure and tool set for reconfigurable computing. QFlow extends TORC with physical netlist modularization techniques, RPM grid generators, a coarse-grain placer and a metadata capturing framework. The entry point to qFlow is an EDIF netlist making front-end tools agnostic; thus, capable of consuming input from graphical and textual tools. Details on the different component of the qFlow framework are described in the next sections.

A. Design Partitioning

The design partitioning process can be done automatically, manually by the designer, or through a designer-guided process. QFlow leaves the process of partitioning the design to the designer. During design partitioning, the designer decides on the boundary separating the invariant and the evolving set, then creates a logical wrapper around the evolving set.

B. Design Jacket Generation

In qFlow nomenclature, the implemented invariant set is referred to as the *design jacket*. Generating the design jacket requires a two-step preparation phase. In Step 1, qFlow parses the logical wrapper (see Section IV-A) interface and generates anchor points for all connections going in and out of the wrapper, in the form of bus macros. Such anchor points come useful when connecting the design jacket to the RPMs in the assembly phase. The wrapper along with the contained evolving logic is then replaced by a corresponding blackbox for implementation purposes. Comes Step 2, qFlow allocates a sandbox using Xilinx *CONFIG PROHIBIT* constraints. The resulting design along with the corresponding constraints are pushed through the Xilinx tools and a fully placed and routed netlist (NCD format) is generated. The resulting netlist excludes any resources in the allocated sandbox region. It is worth mentioning that vendor tools optimization techniques can be applied during design jacket implementation and all timing information is reported back to the user.

C. Evolving Set Implementation

Assumed hierarchical, the evolving set consists of one or more interconnected modules. For implementation purposes, qFlow floorplans, maps and places each module as its own standalone design, allowing qFlow to handle all evolving modules in parallel. The resulting physical netlist is then modularized, i.e., turned into a RPM (NMC format) easily relocatable on the device, and cached for retrieval during the assembly stage. As in Section IV-B, vendor tools optimization techniques can be also applied to each module implementation.

D. Design Assembly

During assembly, the design jacket and the evolving modules are stitched together into the final design. For that purpose, qFlow implements a simulated-annealing coarse-grain placer to handle the placement of evolving modules. The placer reads in the RPM representation of all modules and computes a valid placement for each using a qFlow generated device RPM grid, a two-dimensional representation of all placeable sites on the device. RPM modules are placed by LOCing their reference instance, the instance with respect to which all other instances in an RPM are placed. During placement, unmodified macros are retrieved from cache, whereas updated ones are re-implemented before placement proceeds. All inter-modules and modules to design jacket nets are also created. A successful placement initiates the router stage of the design. Due to its superior quality of results and timing awareness, qFlow leverages Xilinx PAR to perform all its routing, making qFlow a timing-aware framework with timing reporting capabilities. The re-entrant mode of PAR is used to handle the job.

V. Results

Two sets of benchmarks are used to evaluate the flow. The first sixteen designs come from the IWLS 2005 benchmark suite [3], whereas the second set consists of six larger designs, for a total of twenty-two benchmarks. Each design is implemented using Xilinx ISE 13.4 and qFlow for hundred runs. The average of all runs per design is reported in Table II and Table III.

It turned out that all sixteen IWLS 2005 benchmarks are small in size compared to a realistic FPGA design. Therefore we ended up treating them as evolving logic, forcing the framework to place and route them each time the design is assembled. The alternative was to treat them as an invariant logic and make their implementation part of the design jacket with a virtual zero time to implement from iteration to next. Table II presents the compilation times for three flows: Xilinx ISE 13.4, qFLow, and qFlow with an ideal native circuit description (NCD) interface. An *ideal NCD interface* is an interface that takes insignificant time to merge two implemented designs, assuming the two designs have no resource conflicts. The table shows an average speedup of 1.2x using qFlow over Xilinx ISE 13.4.

Three factors come together to cause the moderate speedup reported in this first set of experiments. The first is a slow physical netlist interface. During the assembly stage, qFlow uses Xilinx FPGA Edline, the command line version of FPGA Editor, to merge the different physical netlists. However, FPGA Editor itself is a slow and outdated tool and a replacement is being developed by Xilinx [8]. Secondly, the speedup achieved using the proposed framework is canceled out by the associated large framework overhead given such small designs. For such design sizes, vendor tools excel making the need for acceleration in the first place redundant. Thirdly, by design, all experiments assumed no design jacket re-use, which in turn reduce the speedup using the framework. The last two columns of Table II report a speedup of 1.9x given an ideal physical netlist interface.

The real speedups that qFlow offer is for larger designs that incorporates both invariant and evolving logic. Table III illustrates Xilinx ISE 13.4 and qFlow implementation times for such designs. Each of these designs consists of a design jacket and one or more evolving modules. An average speedup of 4x is attained compared to Xilinx ISE 13.4 for the six designs, and an average 6x speedup for the large Convey HC-1 designs.

TABLE II
FPGA COMPILATION TIMES FOR SIXTEEN IWLS 2005 DESIGNS

Design	**Impl. Time (s)**		**Speedup**	**Impl. Time (s)**	**Speedup**
	ISE	**QFlow**		**QFlow** (Ideal NCD IF)	(Ideal NCD IF)
ac97	146	113	1.3	73	2.0
aes	120	100	1.2	54	2.2
des	123	120	1.0	72	1.7
ethernet	137	178	0.8	127	1.1
fpu	323	284	1.1	231	1.4
i2c	85	80	1.1	47	1.8
mem_ctrl	206	173	1.2	126	1.6
pci	158	109	1.5	60	2.6
sasc	73	78	0.9	46	1.6
simple_spi	75	78	1.0	46	1.6
spi	115	92	1.3	56	2.1
tv80	168	100	1.7	64	2.6
usb_funct	123	130	1.0	83	1.5
usb_phy	80	78	1.0	46	1.7
vga_lcd	133	97	1.4	55	2.4
wb_dma	153	107	1.4	57	2.7

TABLE III
FPGA COMPILATION TIMES FOR SIX SAMPLE DESIGNS

Design	**Platform**	**Impl. Time (s)**		**Speedup**
		ISE	**QFlow**	
Guassian Blur	Xilinx ML 509	350	202	1.73
Edge Detection	Xilinx ML 509	270	165	1.64
ZigBee	Xilinx ML 509	441	155	2.85
1 Vector Addition	Convey HC-1	4084	583	7.01
2 Vector Additions	Convey HC-1	4306	717	6.01
3 Vector Additions	Convey HC-1	4520	909	4.97

VI. CONCLUSION AND FUTURE WORK

Despite their high-performance and modest power consumption, FPGAs are often associated with long place and route times due to their programming model. In this work, we propose an assembly paradigm that leverages the concept of design re-use to speedup the configuration generation process. A framework called qFlow is implemented and despite the limitations of the current vendor tools, an average speedup of up to 6x is achieved for large design. There are a few ways of interpreting this. One is that a designer has the ability to iterate over their design up to six times more over a given period of time. Another is that with a greatly reduced back-end compilation time, a designer is free to explore more design alternatives.

Advancing qFlow to take actions assembly when timing requirements are not met is a potential area to explore. The framework already reports timing information throughout the different implementation and assembly phases. Other areas to investigate consist of quantifying the reduction in design utilization and clock speed when using hierarchical design assembly techniques in general, and the implemented framework in particular. Studying the performance of the tool in more regular designs such as systolic arrays, and design partitioning techniques that maximize re-use, thus decrease compilation times, is yet another venue to check.

VII. ACKNOWLEDGMENT

This work was supported in part by the I/UCRC Program of the National Science Foundation under Grant No. EEC-0642422.

REFERENCES

[1] Partial Reconfiguration User Guide. Technical report, Xilinx Inc., October 2010.
[2] Quartus II Handbook Version 11.1, Volume 1: Design and Synthesis. Technical report, Altera Inc., November 2011.
[3] Christoph Albrecht. Iwls 2005 benchmarks, 2005. `http://iwls.org/iwls2005/benchmarks.html`.
[4] Altera. End markets. `http://www.altera.com/end-markets/end-index.html`.
[5] Peter M. Athanas, J. Bowen, T. Dunham, Cameron Patterson, J. Rice, Matthew Shelburne, Jorge Surís, Mark B. Bucciero, and Jonathan Graf. Wires on demand: Run-time communication synthesis for reconfigurable computing. In *FPL*, pages 513–516, 2007.
[6] Azido. `http://www.azido.net`.
[7] James Coole and Greg Stitt. Bpr: Fast fpga placement and routing using macroblocks. In *Proc. of CODES/ISSS'12: IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, Tampere, Finland, October 2012.
[8] Xilinx Forums. Hard macro external pins deleted when deleting a net, 2011. `http://forums.xilinx.com/t5/Implementation/Hard-macro-external-pins-deleted-when-deleting-a-net/td-p/120124`.
[9] Tannous Frangieh, Athira Chandrasekharan, Suresh Rajagopalan, Yousef Iskander, Stephen Craven, and Cameron Patterson. PATIS: Using Partial Configuration to Improve Static FPGA Design Productivity. In *17th Reconfigurable Architectures Workshop (RAW 2010)*, Atlanta, GA, 2010.
[10] Tannous Frangieh, Richard Stroop, Peter Athanas, and Teresa Cervero. A Modular-Based Assembly Framework for Autonomous Reconfigurable Systems. In *International Symposium on Applied Reconfigurable Computing (ARC 2012)*, Hong Kong, China, 2012.
[11] Paul Glover and Steve Elzinga. Relationally placed macros, 2008. `http://www.xilinx.com/support/documentation/white_papers/wp329.pdf`.
[12] Mentor Graphics. `http://www.mentor.com/`.
[13] Steven A. Guccione, Delon Levi, and Prasanna Sundararajan. JBits: A Java-based Interface for Reconfigurable Computing. In 2nd *Annual Military and Aerospace Applications of Programmable Devices and Technologies Conference*, 1999.
[14] Kate Kelley. Increased Productivity Using Team Design. Technical report, Xilinx Inc., March 2011.
[15] Christopher Lavin, Marc Padilla, Jaren Lamprecht, Philip Lundrigan, Brent Nelson, and Brad Hutchings. HMFlow: Accelerating FPGA Compilation with Hard Macros for Rapid Prototyping. In *Field-Programmable Custom Computing Machines (FCCM), 2011 19th IEEE Annual International Symposium on*, pages 117–124, May 2011.
[16] Christopher Lavin, Marc Padilla, Philip Lundrigan, Brent Nelson, and Brad Hutchings. Rapid Prototyping Tools for FPGA Designs: RapidSmith. In *Field-Programmable Technology (FPT'10). International Conference on*, December 2010.
[17] Chandra Mulpuri and Scott Hauck. Runtime and quality tradeoffs in fpga placement and routing. In *Proceedings of the 2001 ACM/SIGDA ninth international symposium on Field programmable gate arrays*, FPGA '01, pages 29–36, New York, NY, USA, 2001. ACM.
[18] Ali Asgar Sohanghpurwala, Peter Athanas, Tannous Frangieh, and Aaron Wood. OpenPR: An Open-Source Partial-Reconfiguration Toolkit for Xilinx FPGAs. In *18th Reconfigurable Architectures Workshop (RAW 2011)*, Anchorage, AK, 2011.
[19] Neil Steiner, Aaron Wood, Hamid Shojaei, Jacob Couch, Peter Athanas, and Matthew French. Torc: towards an open-source tool flow. In John Wawrzynek and Katherine Compton, editors, *Proceedings of the ACM/SIGDA 19th International Symposium on Field Programmable Gate Arrays, FPGA 2011, Monterey, California, USA, February 27, March 1, 2011*, pages 41–44. ACM, 2011.
[20] Xilinx. `http://www.xilinx.com/`.
[21] Xilinx. Applications. `http://www.xilinx.com/applications/index.htm`.