RAPID FPGA DESIGN PROTOTYPING THROUGH PRESERVATION OF SYSTEM LOGIC: A CASE STUDY

Travis Haroldsen, Brent Nelson, and Brad White

NSF Center for High-Performance Reconfigurable Computing (CHREC)
Department of Electrical and Computer Engineering
Brigham Young University, Provo, UT 84602
Email: trharoldsen@gmail.com, nelson@ee.byu.edu, brad.s.white@gmail.com

ABSTRACT

FPGA designs often contain significant amounts of logic such as a board support package that remains unaltered throughout the design process. However, during normal operation, standard FPGA implementation tools re-implement the entire system, including the unchanged logic, adding to the turn around time of design iterations. Recently, FPGA implementation flows have appeared that allow preserving parts of a previously implemented design. In this study, we evaluate the potential speedups in implementation time achievable through preserving the unchanging portion of a design's implementation. We perform these evaluations using Xilinx Partitions, Xilinx SmartGuide, and the HMFlow rapid implementation tool.

1. INTRODUCTION

In the design of FPGA-based applications there has been a long standing desire for a rapid and interactive design implementation flow. In software development we have come to expect near-instant rapid compile/execute/debug cycles. Once a design has been compiled, FPGAs can be immediately configured for execution and debug, but FPGA compilation times are typically too slow to support a truly interactive development process. Thus, design and debug iterations for FPGA-based systems are often measured in hours rather than seconds or minutes.

FPGA system designs can often be split into two parts which we call the system logic and the user logic. The system logic contains components of the design that are infrequently modified by the designer (if at all). Such components may be cores such as soft-core processors, memory controllers and standard communication interfaces and often may be part of a board support package. The user logic portion of the design consists of the parts of the design that are actively being developed or modified. In this way, there is a rough analogy that can be drawn between software system libraries vs. system logic and user programs vs. user logic.

Importantly, the system logic portion of a design often utilizes the majority of the FPGA resources. However, while the system logic may be unchanged from a previous compilation run, standard FPGA compilation flows typically do not preserve information from previous tool runs. Other studies have shown that using previously compiled components can improve design productivity [1] [2] [3].

The goal of this study was to explore the potential productivity improvements which could be achieved by precompiling the system logic and then reusing it in later compilations. In this study, we looked at three compilation flows which can reuse previous results to potentially decrease runtimes. These included Xilinx Partitions, Xilinx SmartGuide, and a hard macro-based design flow called HMFlow. For each flow, we looked at the compilation speedups achieved when preserving the system logic while making incremental changes to the user logic.

The balance of this paper is as follows. Section 2 first provides background on the three different design flows tested. Section 2.4 follows with a discussion of related work. Section 3 provides a discussion of the designs used for the experiments and the presents experimental results. Finally, Section 4 provides conclusions and suggestions for future work.

2. BACKGROUND AND RELATED WORK

2.1. Background: Xilinx Partitions

Xilinx Partitions [4], or simply partitions, are hierarchical divisions in a design. Typically, partitions are associated with an area constraint to either localize the logic or to force it to a certain area on the chip. Xilinx tools preserve the partition divisions all the way through the implementation process. A subsequent design implementation can then "im-

This work was supported in part by the I/UCRC Program of the National Science Foundation within the NSF Center for High-Performance Reconfigurable Computing (CHREC), Grant No. 0801876.

978-1-4799-0004-6/13/$31.00 ©2013 IEEE

port" implemented partitions from a previous run into the new design.

When importing a partition, the user can set the partition preservation level to *synthesis*, *placement*, or *routing*, allowing control over how much of the partition's previous implementation should be reused. For example, when the preservation level is set to *routing*, the implementation tools will import a fully placed and routed partition, effectively copying and pasting the partition to the same chip location in the new design. When the preservation level is set to *synthesis*, however, only the partition's synthesis results are reused, forcing the partition to be newly placed and routed into the new design.

To use partitions, the designer tags portions of the design hierarchy, each such portion forming a partition. An optional area constraint can be attached to each of these. The initial place and route then creates each partition's implementation. Instead of implementing each partition separately, it is recommended that all partitions in a design be present for the initial implementation run. The advantage of this is that the implementation tools can better optimize the placement of ports connecting the different partitions and achieve a better global routing result. Also, while in theory one could envision assembing a design solely by importing a collection of partitions from multiple designs, there are often routing conflicts between adjacent partitions, especially if they are closely placed to one another. For this reason, partitions are typically pulled from only a single previous design run.

Xilinx guidelines recommend partitions principally for preserving timing critical sections of a design or for dividing a project so team members can work on individual sections independently of one another. Using partitions for productivity improvements (the focus of this work) seems to only be of secondary concern, but is possible if critical paths are preserved [5].

2.2. Background: Xilinx SmartGuide

Xilinx SmartGuide is an alternative to partitions, such that one or the other may be used in a design's compilation, but not both. SmartGuide, commonly known as "guide files," uses a previous implementation to guide a subsequent implementation. In other words, a previously technology-mapped, placed, and/or routed NCD file may be input to either MAP or PAR to guide technology-mapping, placement and/or routing. Specifically, SmartGuide compares the "guide" implementation to the new design and preserves previous decisions made by the tools on unchanged portions of the design.

An advantage of using SmartGuide is that, unlike the planning required on the part of the designer to use partitions, there is no preparation required to use SmartGuide. When re-implementing a design, the user need only supply a reference to a previously implemented design to be used as a guide for the new implementation.

Our experiments with various designs suggest that SmartGuide can do well with small incremental changes to a design such as changing the value of a constant, adding a register, or even modifying a VHDL architecture, depending on its size. While the re-implementation cost savings can be significant, the time reduction seems to be quite design-dependent in our experience. Xilinx suggests the use of SmartGuide when changing less than 10% of a design [6].

2.3. Background: HMFlow

HMFlow [1] is a rapid FPGA implementation flow designed to improve FPGA design implementation times through the use of pre-assembled hardware modules called *hard macros*. A hard macro is a fully synthesized/placed/routed subdesign, similar in some ways to a partition. However, a hard macro can be relocated around the FPGA fabric by the HMFlow tool.

HMFlow is based on the RapidSmith tool set [7] [8] and thus works outside the normal Xilinx tool flow. RapidSmith operates on XDL files rather than .ncd files — the XDL language [9] is an open design format provided by Xilinx in the ISE tool suite and is a textual representation of much of the information found in a .ncd file. Designs can be converted between .ncd and XDL using the *xdl* command line tool from the ISE tool suite.

A hard macro is created by running the conventional Xilinx tool flow with appropriate constraints to control its layout and shape [10]. It can thus be created from any arbitrary RTL source (HMFlow currently supports the creation of hard macros from either VHDL or SystemGenerator source). Once implemented by the Xilinx tools, the hard macro design is then converted into the Xilinx XDL format and further processed for use by HMFlow.

To compile a complete design composed of hard macros, HMFlow first searches its cache for each hard macro required by a design. If the hard macro for a block does not exist in the cache, HMFlow will generate it using the process described above and store it in its macro cache. Next, HMFlow assembles the hard macros required for the design and then places and routes the resulting ensemble of blocks using its own custom built placer and router. The result is an XDL representation of the fully placed and routed design which is then converted from XDL back to NCD and a bitstream is generated.

For this experiment, HMFlow was extended to include support for "system macros" to contain all of the non-changing system logic in a design. A system macro is similar to any other hard macro in that is is a fully placed and routed circuit module. However, since it typically contains I/O sites it will not be relocatable around the FPGA fabric. Thus, a system macro-based design will consist of a statically-placed

system macro with an open area available in the fabric for rapidly inserting various user design modules.

2.4. Background: Other Related Work

A variety of other research projects have looked at block-based design techniques, specifically intended to reduce implementation time. Here we describe two that deal with pre-compiled circuit building blocks.

2.4.1. qFlow

qFlow, an incremental compilation technique developed at Virginia Tech [3], also uses hard macros to realize compilation speedups. Built on the TORC toolset [11], this flow is similar to HMFlow in a number of ways. First, the user manually partitions the design into invariant and evolving logic sets. Then the invariant logic is placed and routed using the Xilinx tools with placement constraints, leaving an open area on the FPGA called the "sandbox." The evolving logic is then implemented as one or more macros (represented in the Xilinx tools as .nmc files). Finally, both the invariant and evolving logic are placed and routed together with the evolving logic being placed into the sandbox area. For subsequent changes to the evolving logic, only the evolving logic needs be re-implemented.

One key difference is qFlow's use of the Xilinx tool chain for placement and routing as well as its use of .nmc files for representing hard macros. In contrast, HMFlow does its design assembly outside the normal Xilinx tool flow using the RapidSmith tool set and an XDL circuit representation. Also, HMFlow employs its own custom-built place and route tools, injecting its finished designs back into the Xilinx tool flow only for bitstream generation.

2.4.2. BPR

Block Place and Route (BPR), developed at the University of Florida, constructs a design using coarse-grain cores [2]. Unlike HMFlow and qFlow, a given core is not turned into a relocatable hard macro, but rather numerous versions of each block are placed and routed (at different locations in the FPGA fabric) and stored in a database. At assembly time, one version of each block is selected to give the desired system layout and then routed together.

2.4.3. Partial Reconfiguration Flows

A variety of design flows have been described to take advantage of the partial reconfiguration capabilities of FPGAs [12] [13] [14], and which have some similarities with rapid design flows. For example, GoAhead [13], divides a design into static and partially reconfigured modules and then floorplans them into specific areas of the device. It provides for the independent implementation of the static and partial modules from one another. It also supports the relocation of partial modules, the elimination of bus macros, and the creation of hierarchical PR regions. The key difference from this work is that PR flows are optimized to support runtime reconfiguration, while this work targets rapid development, specifically the reduction of CAD tool runtime.

3. EXPERIMENTS

The system organization chosen for our experiments consists of a design with two parts. The first is called the system logic, contains the majority of the design, and rarely changes. The second is called the user logic and contains the part of the design which is changed regularly. The experiments of this section were designed to analyze the FPGA compilation runtime improvements that might result when this design is processed using Xilinx Partitions, Xilinx SmartGuide, and HMFlow. For each flow, we created an initial placement of the system logic, made a small design change in the user logic (about 1% of the design), and rebuilt the design while re-using the placed and routed system logic from before. All experimental times given are averages of 100 runs.

3.1. Example Design

Our benchmark for this study was a video filtering system (see Figure 1). The user logic portion of the design consisted of a set of video filters that could be pipelined together in any combination. The base design we used contained four such filters. The filters contained connections to a Processor Local Bus found inside the system logic. The remainder of the circuit was the system logic, a Xilinx EDK-generated design which contained two Xilinx Microblaze Processors, a DDR2 Memory Controller, two large image rotation peripherals, as well as other IO peripherals and busses.

The system was implemented on a Xilinx xc5vlx330 chip (the largest Virtex 5 part). The system logic comprised 95% of the design. Together, the system logic and user logic utilized 55% of the chip's resources (see Table 1). We purposely chose this level of utilization since prototyping is focused more on implementation speed than device utilization. The 55% utilization means the tools have ample area to work within and makes meeting timing constraints easier.

Partition	Slices	% of Chip
System	27,221	52.5
User	1,215	2.3

Table 1. Chip Utilization of Large Design

All of our experiments used the Xilinx ISE 14.2 Tool Suite. As a baseline, the design was compiled using the

Flow	XST	NGD	MAP	PAR	Total
Standard Xilinx Flow	0:19:22	0:01:33	0:24:22	0:09:40	0:54:56
Incremental Synthesis	0:02:20	0:01:52	0:25:24	0:10:10	0:39:45

Table 2. Standard Xilinx Flat Flow Runtimes (H:MM:SS)

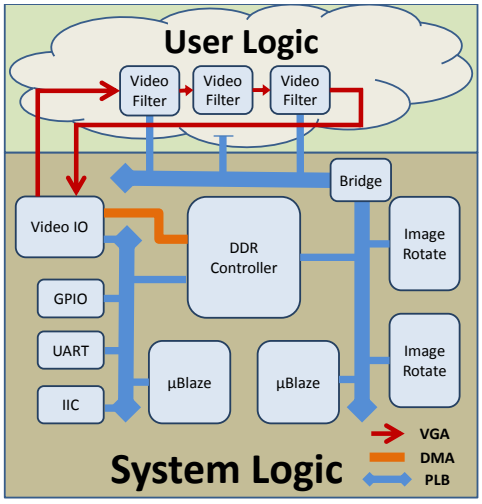

Fig. 1. Frame Buffer Benchmark System

standard Xilinx flow which flattens and completely rebuilds the design for each implementation run. Table 2 shows the runtimes of the standard Xilinx tool flow for this design as well as a re-implementation that uses an incremental synthesis flow to bypass resynthesizing the system logic. All speedups in the remainder of this paper are in comparison to the Standard Xilinx Flow runtimes (row 1).

3.2. Design Considerations

Early experiments building the system logic of Figure 1 pointed out two considerations when partitioning and floorplanning the design. The first consideration was the floorplanning of the IO. An early floorplan contained IO that was placed at locations dictated by the targeted PC board. The result was that IO was scattered across the chip making floorplanning of the system partition difficult and resulting in cross-chip routes. This highlighted the importance of properly locating IO. For this experiment, it was assumed that the board IO placement was determined with the rapid-prototyping flow in mind and IO pin locations were limited to the area allocated for the system logic.

The next consideration was ensuring connections between the system and user logic could meet timing, even when the system logic was built without knowing where the connecting user logic would ultimately be located. To mitigate the resulting issue of long wiring delays, registers were placed on all signals crossing boundary between the system and user logic. An alternative would be a careful floorplanning of pin locations on both the system and user logic to minimize wire length (an interesting topic for future work).

3.3. Experiments With Partitions

3.3.1. Procedure

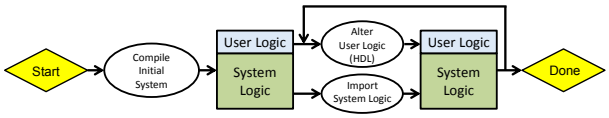

Fig. 2. Partition-based System Incremental Flow

Figure 2 shows the partitioned-based incremental flow we used. The first step of the flow was floorplanning, the goal being to leave enough space for user logic to maximize the possibility of being able to rapidly implement that user logic while still meeting timing.

In our floorplan, the system logic occupied the lower 75% of the chip and the user logic resided in the remaining upper 25% of the chip. This gave the system logic about a 75% utilization rate within its boundaries (within the Xilinx suggested guidelines) and a relatively large area for the user logic. We also explored allowing the user logic to overlap the system logic space, but this had little impact on the results.

After the design was floorplanned, it was ready for its initial implementation run. The purpose of the initial implementation was to build the system logic partition. During this run, the user logic was also implemented into its own partition to hopefully provide representative locations for the signals connecting the system logic to the user logic. The subsequent implementations described below, however, only reused the system partition.

The remaining experiments focused on the iterative loop of Figure 2. These experiments consisted of rebuilding the design while reusing some previously implemented partitions and re-implementing other partitions.

3.3.2. Results

The runtimes of our experiments are shown in Table 3. The times for the initial build only include runs that met timing constraints — when using partitions only 49% successfully met the design's timing constraints as opposed to 100% of the builds meeting timing constraints when partition were

Flow	XST	NGD	MAP	PAR	Total	Speedup
Initial Build	0:21:15	0:02:05	2:39:20	0:22:44	3:05:21	0.29x
Import System	0:02:20	0:02:05	0:19:29	0:09:21	0:33:15	1.65x
Import All Partitions	-	-	0:17:40	0:09:17	0:26:57	2.04x

Table 3. Partition-based Incremental Flow Runtimes (H:MM:SS)

Changes	XST	NGD	MAP	PAR	Total	Speedup
Add a filter	0:03:40	0:01:47	0:15:31	0:07:26	0:27:56	1.97x
Remove a filter	0:01:50	0:01:52	0:15:18	0:07:17	0:26:18	2.09x
Replace a filter	0:01:50	0:01:52	0:15:18	0:07:17	0:26:18	2.09x
No change	0:02:28	0:01:46	0:14:40	0:06:57	0:25:51	2.13x

Table 4. Xilinx SmartGuide Flow Runtimes (H:MM:SS)

not used. The successful initial builds shown in row 1 of Table 3 also took 3.4 times longer to implement than the standard flow implementation. In our experience, this is not an uncommon occurence with partitions and seems more to do with the addition of area constraints to the partitions rather than the creation of partitions themselves. However, this slowdown may be of minor consequence since the objective of this system-based incremental flow is to only run the initial build once.

The second row of Table 3 (Import System) shows the runtimes for iterations through the inner loop of Figure 2 where the previously implemented system partition was imported into the design but a change was made to the user logic forcing a rebuild of the user partition. Inside this loop, the pre-implemented system partition is imported into the new design, the new user logic is built (synthesized, placed and routed) and the system and user partitions are routed together. In our benchmark, we saw a 65% improvement over the standard Xilinx flow in these experiments.

The third row of Table 3 (Import All Partitions) shows the runtime when both the user and system logic were imported from previous runs. This can be viewed as a lower-bound on the tool runtime for partitioned designs of this size since the tools need only route the nets crossing the system/user partition boundary. As shown in the table, this gives a 2× speedup over the standard tool flow.

3.4. Experiments With SmartGuide

3.4.1. Procedure

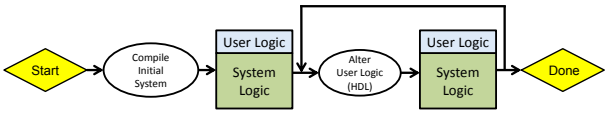

Fig. 3. SmartGuide-based System Incremental Flow

Unlike partitions and HMFlow, SmartGuide requires no additional floorplanning of a design – the only change to the standard toolflow is the inclusion of a previously routed design, its associated guide (.ngm) file, and the "-smartguide" flag when calling MAP or PAR in the standard tools.

To measure improvement with SmartGuide, our experiments consisted of making a change to the original design and re-implementing the changed design using SmartGuide. Because Smartguide modifies a pre-placed and routed design as opposed to rebuilding the design as in Partition flow, different types of changes may have a wider effect on the implementation runtime than occur in partitions and HMFlow. To address the greater variability, we tested Smartguide with a set of changes that we felt addressed common modifications that might be made to a design.

The list of changes included: (1) Adding an image filter, (2) Removing an image filter, (3) Replacing one filter with another, and (4) No change. Each of these changes except for "No change" affected about 1% of the total design.

3.4.2. Results

Table 4 shows the runtimes for each change based on approximately 100 runs. Incremental synthesis (see row 2 of Table 2) was used to avoid resynthesizing the sytem logic reducing the XST time. The speedup values given are with respect to the Standard Xilinx Flow row from Table 2. All runs for each modification successfully met the design timing constraint.

Interestingly, there was little variation between the average run times – it would seem that re-implementing the design with no changes takes about as long as any of the changes we tried. This suggests that for this particular design there is a 25 minute overhead when using SmartGuide. Still, for this design SmartGuide provides about a 2x improvement over a standard tool run, of the same order as the improvements provided by partitions.

Flow	Macro Generation/Load	Stitch	Place	Route	Total	Speedup
Initial Build	3:04:40	0:00:18	0:00:00	0:01:20	3:06:18	0.29x
Import System	0:05:57	0:00:18	0:00:00	0:01:20	0:07:35	7.24x
All Macros Prebuilt	0:00:08	0:00:18	0:00:00	0:01:20	0:01:46	31.09x

Table 5. HMFlow: Single User Hard Macro Runtimes (H:MM:SS)

Flow	Macro Generation/Load	Stitch	Place	Route	Total	Speedup
Initial Build	3:15:41	0:00:38	0:00:00	0:00:59	3:17:18	0.28x
Import System	0:04:58	0:00:38	0:00:00	0:00:59	0:06:35	8.34x
All Macros Prebuilt	0:00:08	0:00:38	0:00:00	0:00:59	0:01:45	31.39x

Table 6. HMFlow: Four User Hard Macro Runtimes (H:MM:SS)

3.4.3. Observations on SmartGuide vs. Partitions

A unique aspect of SmartGuide is that it does not distinguish between user and system logic. This means that with SmartGuide, changes can be made anywhere in the design without necessarily requiring a long re-implementation. In contrast, our use model for partitions above assumes a static collection of system logic with changes being made principally to the user logic. With our partitions experiment, a change to the system partition would require a long re-implementation (close to 3 hours for our example).

3.5. Experiments With HMFlow

3.5.1. Procedure

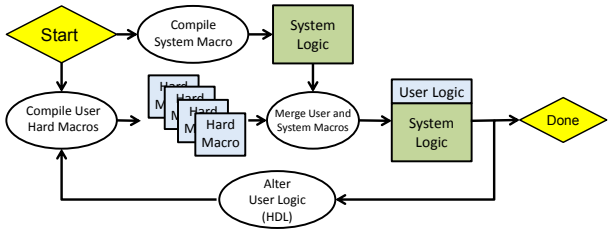

Fig. 4. HMFlow-based System Incremental Flow

For the HMFlow experiments, we used the same floorplan used in the partition flow, with the bottom 75% of the chip allocated for the system logic and the remaining 25% for user logic. Figure 4 shows the steps used in the HMFlow experiment. The first step was to generate the hard macros, including both the system logic macro as well as the user logic macro(s), shown in the upper left of the figure. The resulting macros, shown in the figure as the blocks labelled "System Logic" and "Hard Macro" were all then cached for later reuse. The next step was to place and route the system macro and user hard macros into a finished layout (right side of figure). Then, for repeated design iterations around the loop of Figure 4, the user logic was modified, the required hard macros were either loaded from the cache or generated (if needed), and the new collection of macros placed and routed into a design.

3.5.2. Results

In our experiments, we evaluated both creating a separate hard macro for each of the four filters used in the testing and placing all user logic into a single hard macro. The results are presented in Tables 5 and 6 respectively. The speedup values given are with respect to the Standard Xilinx Flow row from Table 2.

Row 1 of the tables (*Initial Build*) shows the runtime for generating both the system macro and all needed hard macros as well as assembling an initial design from them. As with the partition flow above, a portion of the initial system macro builds failed to meet timing and the runtimes of those implementations have been excluded. Additionally, as with the Xilinx partitions initial build, this step should be a one time occurence.

Row 2 of the tables (*Import System*) assumes one filter was modified and had to be regenerated while the previously implemented system macro was simply imported into the design. In the case of the single user hard macro tests, this meant the entire user hard macro had to be regenerated. In the case of the tests with four individual user hard macros, this meant only one of the macros had to be rebuilt.

Finally, in row 3 of the tables (*All Macros Prebuilt*), all of the required macros already existed in the cache and so only final design assembly (placement of the user macro(s) and routing to the system macro) was required. This can be considered a lower bound on implementation time for HMFlow. For designs constructed solely from pre-defined building blocks it does represent an achievable speedup.

HMFlow performed well in our experiments. When hard macros needed to be generated we saw runtimes for placement and routing of 6-7 minutes or 2 minutes when all macros

previously existed. This shows that significant speedups are possible with a system macro-based flow.

4. CONCLUSION

For this particular design and iterative design flow, our experiments showed modest improvements with Xilinx Partitions and Xilinx SmartGuide tool runs time while custom assembly approaches such as HMFlow showed great potential for speeding up implementation time. This work suggests a number of avenues for future work. Different relative sizes of the system vs. user logic sections of a design will undoubtedly result in different speedups. Also, as was mentioned in Section 3, the I/O pin placement within a system macro affects the resulting design implementation. Future work could explore the interactions between FPGA and PC board I/O pin placement for such a flow since different I/O placement will lead to different feasible floorplans for a system+user logic structure as described here. Finally, as described in Section 2.4, a variety of custom flows that operate both inside and outside the vendor tools have been proposed, and new vendor tools such as Vivado promise to provide new possibilities for rapid design flows. These all represent fruitful areas for further investigation.

5. REFERENCES

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

[2] J. Coole and G. Stitt, "BPR: Fast FPGA Placement and Routing Using Macroblocks," in *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, ser. CODES+ISSS '12. New York, NY, USA: ACM, 2012, pp. 275–284. [Online]. Available: http://doi.acm.org/10.1145/2380445.2380491

[3] T. Frangieh, "A Design Assembly Technique for FPGA Back-End Acceleration," Ph.D. dissertation, Virginia Polytechnic Institute, Blacksburg, Sep. 2012. [Online]. Available: http://scholar.lib.vt.edu/theses/available/etd10082012-021855/unrestricted/Frangieh_T_D_2012.pdf

[4] Xilinx. (2001, Mar.) Hierarchical Design Methodology Guide. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_1/Hierarchical_Design_Methodology_Guide.pdf

[5] K. Kelley. (2011, Feb.) Hierarchical Design Using Synopsys and Xilinx FPGAs. White Paper. Xilinx. [Online]. Available: http://www.xilinx.com/support/documentation/white_papers/wp386_Hierarchical_Design_Synopsys_Xilinx.pdf

[6] Xilinx. (2007, Jun.) Incremental Design Reuse with Partitions. [Online]. Available: http://www.xilinx.com/support/documentation/application_notes/xapp918.pdf

[7] C. Lavin, M. Padilla, J. Lamprecht, P. Lundrigan, B. Nelson, and B. Hutchings, "RapidSmith: Do-It-Yourself CAD Tools for Xilinx FPGAs," in *Proceedings of the 2011 21st International Conference on Field Programmable Logic and Applications*, ser. FPL '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 349–355. [Online]. Available: http://dx.doi.org/10.1109/FPL.2011.69

[8] C. Lavin, M. Padilla, P. Lundrigan, B. Nelson, and B. Hutchings, "Rapid prototyping tools for FPGA designs: RapidSmith," in *Field-Programmable Technology (FPT), 2010 International Conference on*, Dec. 2010, pp. 353–356.

[9] C. Beckhoff, D. Koch, and J. Torresen, "The Xilinx Design Language (XDL): Tutorial and use cases," in *Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2011 6th International Workshop on*, Jun. 2011, pp. 1 –8.

[10] J. Lamprecht and B. Hutchings, "Profiling FPGA Floor-Planning Effects on Timing Closure," in *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, Aug. 2012, pp. 151–156.

[11] 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*, ser. FPGA '11. New York, NY, USA: ACM, 2011, pp. 41–44. [Online]. Available: http://doi.acm.org/10.1145/1950413.1950425

[12] A. Sohanghpurwala, P. Athanas, T. Frangieh, and A. Wood, "OpenPR: An Open-Source Partial-Reconfiguration Toolkit for Xilinx FPGAs," in *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, 2011, pp. 228–235.

[13] C. Beckhoff, D. Koch, and J. Torresen, "GoAhead: A Partial Reconfiguration Framework," in *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*, 2012, pp. 37–44.

[14] D. Koch, C. Beckhoff, and J. Teich, "ReCoBus-Builder - A novel tool and technique to build statically and dynamically reconfigurable systems for FPGAS," in *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*, 2008, pp. 119–124.