IMPACT OF HARD MACRO SIZE ON FPGA CLOCK RATE AND PLACE/ROUTE TIME

Christopher Lavin and Brent Nelson and Brad Hutchings
NSF Center for High-Performance Reconfigurable Computing (CHREC)
Department of Electrical and Computer Engineering
Brigham Young University, Provo, UT 84602
Email: nelson@ee.byu.edu, hutch@ee.byu.edu, chrislavin@byu.net

Abstract—**Hard macros are completely placed/routed elements that are treated as primitives and that are relatively placed as a single element. A system composed of such macros consists of many fewer effective primitives and nets and as such can be placed and routed much more quickly. Prior work in this research area dealt with small, general-purpose macros such as 16-bit registers, adders, etc., and demonstrated that place/route time could be reduced by an order of magnitude with a corresponding 3-4X reduction in clock rate. In this work, much larger hard macros are developed such as mixers, soft-core processors, FFTs, etc., and the use of these larger macros is shown to further reduce place/route time by an additional 2.5-4X, for a total of a 30-40X reduction in compile time. Clock rate is also improved, relative to earlier work, by an additional 60-70%.**

I. Introduction

FPGA place and route cycles that consume hours or days are a major impediment to in-system verification. The basic FPGA design flow in use today was largely adopted from the decades-old ASIC design flow consisting of HDL coding followed by simulation. While this approach is entirely appropriate for a non-existent, yet-to-be fabricated device, it completely ignores the inherent advantage of the FPGA as an available device that can be used to accelerate and improve the verification process. Simulation is, of course, an essential tool that provides observability and faster compilation time (relative to place/route) though it runs about 1,000,000 times slower than execution on the FPGA device. However, simulation can never replace the process of actually inserting an FPGA into a system and running it with actual data in real-time. Actual insertion of the FPGA into the operating environment is the only way to verify that the design works as intended and lengthy compile cycles make regular in-system verification very difficult or nearly impossible.

A variety of methods have been proposed in the past for accelerating hardware debug and verification times. One such approach is to use pre-compiled circuit modules to reduce the effort required for synthesis, placement, and routing. For IC design, the use of pre-designed and compiled "cell libraries" is a common approach. These pre-compiled modules can dramatically reduce build time by eliminating repeated re-compilation of common building blocks.

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.

HMFlow[1] is a FPGA design tool that employs pre-built circuit modules to reduce implementation time. The pre-compiled modules used by HMFlow are referred to as "hard macros" and consist of previously synthesized/mapped/placed/routed building blocks that are stored in a library for later use during rapid design implementation and assembly. The hard macros initially used in HMFlow were relatively small and consisted of simple general-purpose computational elements such as adders, registers, multiplexers, etc. As reported, the HMFlow system[1] achieved an order of magnitude reduction in compilation time but resulted in circuits with clock rates often significantly lower than those produced by vendor FPGA implementation tools.

This work focuses primarily on improving clock rates while preserving or further reducing compilation time. The overall approach taken in this work is to increase the size of the hard macros used in the flow. Rather than use small general-purpose hard macros, this effort investigates the use of much larger system-level sized hard macros such as FIR filters, FFTs, DDS/mixers, micro-controllers and so forth. The hypothesis is that larger hard macros may improve clock rate and potentially further reduce implementation time because: (1) large macros contain far more routing and thus provide an opportunity to preserve the computing effort used to achieve timing closure and, (2) designs will consist of fewer hard macros and require correspondingly less effort to place and route them.

This effort tests this hypothesis by answering the following primary research questions:

1) What impact does the use of large hard macros have on tool runtime? That is, how much does the resulting reduction in number of objects to be placed and routed help when assembling designs from pre-defined circuit blocks?
2) Can large hard macros truly encapsulate sufficient timing closure effort to significantly improve the final clock rate of macro-based circuits?
3) What problems do large hard macros create for a CAD

tool flow like HMFlow in terms of placement-based performance variation of the final circuits or the place-ability and route-ability of the final circuits?

The rest of the paper is outlined as follows: In Section II we describe prior efforts to accelerate FPGA compilation including that of HMFlow, which this work is based on. In Section III we present and describe the new hard macros and design approach employed in this work. In Section IV a series of benchmark circuits and the resulting performance achieved is presented and analyzed. In Section V we conclude and describe future work.

II. Background and Related Work

A. HMFlow Background

This work is based on HMFlow, a custom-built CAD tool flow based on the use pre-compiled circuit building blocks called "hard macros" to rapidly assemble final circuits [1]. A block diagram of HMFlow is given in Figure 1.

The flow begins by processing designs created using Xilinx System Generator (which generates .mdl files). In the Design Parser & Mapper, each block in the design and its corresponding hard macro are identified. If the hard macro does not exist in the hard macro cache, the mapper invokes the hard macro generator to create one and to store it in the cache.

Once all of the hard macros have been created or retrieved from the cache, they are given to the design stitcher, which will "stitch" all of the hard macros together logically and then insert I/O buffers and clock generation circuitry. The design is then passed to the HMFlow hard macro placer and router tools and ultimately exported as a final placed and routed implementation.

HMFlow operates outside the normal Xilinx tool flow by operating on XDL files rather than .ncd files. The XDL language 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. Additionally, Xilinx also provides detailed physical information on each of its FPGAs in the form of XLDRC files. Converting designs from the Xilinx proprietary .NCD file format to/from XDL files is done using the ISE *xdl* command and can be done essentially anywhere in the Xilinx tool chain — before placement, before routing, or after both placement and routing as shown in Figure 2.

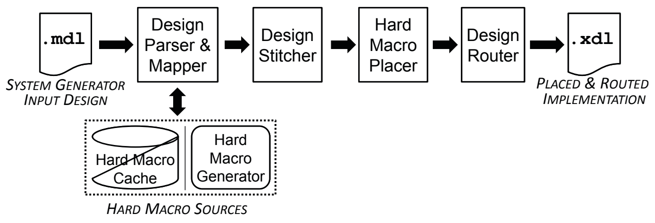

Fig. 1: Block Diagram of HMFlow

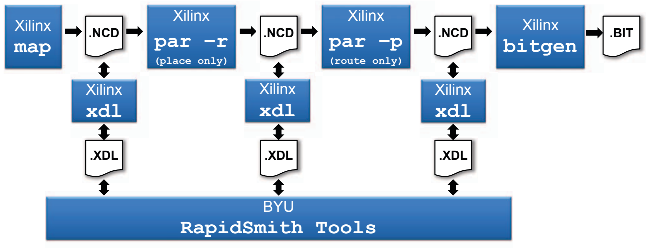

Fig. 2: Illustration of where XDL interfaces with the Xilinx tool flow and how RapidSmith interacts with XDL.

HMFlow is built on top of RapidSmith [2], an XDL processing and manipulation CAD tool framework. Given a design in XDL format and an XDLRC description of an FPGA, RapidSmith provides a rich set of API calls for creating, modifying, placing, and routing circuit elements in the FPGA as well as for importing designs from XDL files and for creating XDL files as output.

In HMFlow, a hard macro is originally created using the conventional Xilinx tool flow. First, the logic for the hard macro is synthesized and then placed and routed into a small completed design, with constraints being used to restrict its circuit elements to a small rectangular area in the FPGA. Then it is converted to XDL, the IOB's removed (which were automatically inserted during the Xilinx implementation process), and the finished macro (represented as an XDL module) placed into the HMFlow macro cache for later use.

As previously reported [1], each System Generator design element was turned into a hard macro, a typical hard macro being a 32-bit adder/subtracter, a 16-bit comparator, or even something as simple as a 5-bit AND gate. A variety of benchmarks were used to evaluate the approach; a representative design, a trellis decoder, consisted of 17,000 slices, 60 BRAMs and 50 DSP48 units and was implemented using about 1,300 hard macros.

B. Related Work

The approach taken in this work is to use pre-compiled blocks called hard macros, however, other techniques such as bitstream cores, macroblocks, virtual fabrics and a tool called ReCoBus also demonstrate ways in which intermediate design information can be reused to reduce compilation time.

Horta and Lockwood [3] demonstrated the creation of bitstream-based re-locatable cores that are quite similar in nature to hard macros. Similar efforts are reported in [4] where bitstream hard cores were used in a network-on-chip to provide accelerated logic emulation and prototyping. Unfortunately, bitstream hard cores must reside between restrictive configuration boundaries, must be augmented with matching bus-macro interfaces and can be difficult to construct because bitstream formats are typically proprietary.

ReCoBus [5] can create bus-based systems that can be loaded using partial reconfiguration. ReCoBus hard macros consist of user logic and an interface to the ReCoBus system bus and are converted to special partial bitstreams so they can be swapped in and out of the FPGA at run-time. Full systems can be rapidly constructed with the ReCoBus bitstream linker.

Intermediate virtual fabrics[6] implement a domain-specific fabric on top of a conventional FPGA. These fabrics accommodate macroblocks which are placed and routed quickly onto the fabric. This technique is effective if the intermediate fabric has already been built for a particular application and is a close match to the domain of interest. Coole et al. [6] claim an average place and route speedup of 554×. However, the technique is only effective to the extent that the intermediate fabric is reused. If no available intermediate fabric matches your application requirements, you must design and implement a new fabric, a time-consuming step that reduces the impact of fast compilation.

The prior work closest to this effort is Frontier[7], a placement tool that utilized macroblocks and floorplanning to accelerate placement. Macroblocks are similar to HMFlow macros; they were precompiled and some of them could be relatively placed. Frontier decomposes the FPGA into a set of placement bins of equal size; macroblocks are grouped into clusters and are initially assigned to placement bins. Placement quality is improved by swapping clusters between bins and a low-temperature annealing process can be employed to further improve the placement. Frontier accelerated placement by 17×; this results in an overall acceleration of 2.6× for the overall place and route process.

The major difference between HMFlow and Frontier is that while HMFlow hard macros contain routing and placement information, Frontier macroblocks only contain placement information. Once the macrocells are placed by Frontier, vendor tools must completely reroute all nets. Because HMFlow hard-macros contain both internal hard-macro routing, they reuse significantly more computational effort. As such HMFlow can significantly reduce run-times for both placement and routing. In addition, preserving routing means that much of the computational effort to close timing is also preserved and this can lead to a higher quality result.

III. Approach

This effort seeks to measure the impact of larger hard macros on both clock rate and compilation run-time. Larger hard macros potentially benefit HMFlow in two major ways. First, larger hard macros contain a much higher percentage of the total routes in the design thereby preserving timing-closure information in the form of routed nets that can be re-used with each instantiation of the hard macro. Second, designs composed of larger hard macros can be more rapidly placed as they consist of significantly fewer blocks.

For example, the frequency estimator benchmark used to demonstrate HMFlow with smaller hard macros[1] contained 11,226 unrouted nets external to the hard macros. This constitutes 66% of all nets in the design (nets inside and outside the hard macros). These unrouted nets are those nets that are not contained within the hard macro and are used to connect the hard macros to each other and to system I/O. A version of the frequency-estimate benchmark implemented using much larger hard macros has only 854 unrouted nets (only 5% of total nets) before the design reaches the HMFlow router, resulting in a significant reduction in work for the HMFlow router.

The large hard-macro version of the frequency-estimator also consisted of many fewer macros than its small hard-macro cousin. This same frequency-estimator benchmark consisted of 757 small, general-purpose macros [1]. A version employing large hard macros contains only 31 hard macro instances to be placed, a 24× reduction that will significantly reduce run-time for the placer.

A. Preliminary Work

Two major questions needed to be answered regarding the feasibility of large hard macros. They include:

1) *Are hard macros completely relocatable?* If a large hard macro is created and timing verified to run at clock rate X at one location, will it be able to run at the same or similar rate at another location on the FPGA fabric?
2) *What is the best size, shape and aspect ratio for hard macros?* As reported in prior work[1] the hard macros were simple enough that their layouts were typically vertical to take advantage of carry chains and the like. For larger hard macros there is much more flexibility for choosing different shapes and form factors when creating the hard macros.

To answer these questions, a series of experiments were conducted. The first set of experiments (named Experiment #1) was performed to measure the timing variability of hard macros as they are placed at different locations across the FPGA fabric. To do this, two different hard macros containing a reasonable set of internal routes were created. The first hard macro created was a 21×21 bit LUT-based multiplier on a Virtex 4 SX35 FPGA. Relative to the FPGA fabric size, the hard macro was small, but did contain several timing sensitive routing paths that could be accurately measured and variations in its timing behavior could easily be detected.

A separate implementation containing a unique placement for each valid location of the hard macro was then created. For each of the resulting 400 legal unique placements of the hard macro, the implementation was completed and the timing results were measured. Placement affected timing in two different ways.

The first is shown in Figure 3 where the Z-axis represents the amount of delay (in nanoseconds). The regular extra delays that occur in the Y direction correlate very well with the locations of the horizontal clock branches found in the FPGA, suggesting that longer wire paths are required

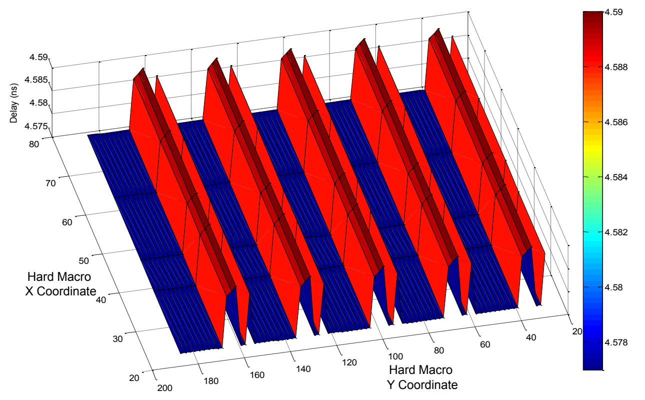

Fig. 3: Delay of a Path Within a 21×21 Bit LUT-multiplier Hard Macro Placed in a Grid of 400 Locations on a Virtex 4 SX35 FPGA

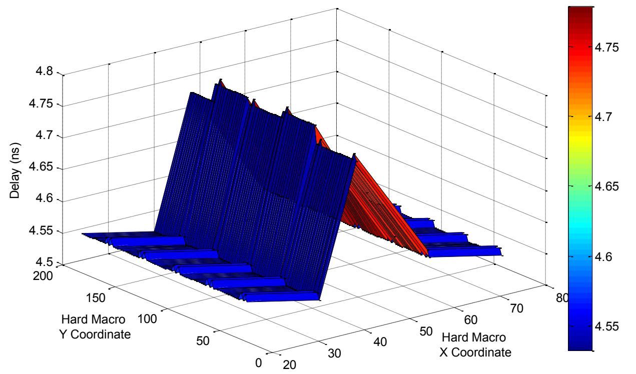

Fig. 4: A More Severely Impacted Path Caused by a Hard Macro Straddling the Center Clock Tree Spine of the FPGA

as signals traverse those horizontal clock branches. The deviations, however, are quite minimal ($\sim$ 10 ps).

A second type of timing deviation (in the X direction) was also found and is shown in Figure 4 (note the different perspective compared to Figure 3). Here the timing deviation is much more significant ($\sim$ 250 ps) and is due to the hard macro straddling the center clock distribution column of the FPGA. If examined closely, the same ridges present in Figure 3 can be seen, although they are dwarfed by the major ridge down the center column of the FPGA. This delay is more problematic but can easily be avoided by simply disallowing hard macro placements that straddle the center column of the chip.

A second hard macro (consisting of a PicoBlaze core) was also created and similar experiments conducted but for Virtex 5 devices (without allowing the hard macro to ever straddle the center column). The results were almost identical with timing deviations in the Y direction ($\sim$ 10 ps) due to the horizontal clock branches.

A second set of experiments (named Experiment #2) was performed to determine the impact of shape (aspect ratio) and density on the performance of the resulting hard macros. These experiments were detailed in [8] and [9] and only a short summary given here.

The different hard macros used in those experiments consisted of the following: a 1024 point FFT, an 18×18 bit LUT-based multiplier, 128-tap 18-bit FIR filter, a double-precision quadratic solver, a PicoBlaze and a MicroBlaze. Two different characteristics of each hard macro was varied. The first variable was the width to height (aspect ratio) of the macro. The second was the density of the macro which was the ratio of total area allocated versus the number of slices and other resources required for the macro's logic. The candidate hard macros were then implemented using a variety of clock constraints. The data indicate that most macros are aspect-ratio agnostic and that about 20% extra area should be allocated when determining the shape and area for the hard macro.

From these experiments we concluded that hard macros can be generated in a way that maximizes their place-ability and should not be placed over the center column of the chip. Thus, in the experiments of the next section the large macros required were generated with roughly square shapes (with variations from this dictated by the particular macro's use of DSP48 and BRAM blocks).

IV. Experimental Results

To investigate the effects of using large hard macros, six large benchmark designs were chosen and re-fashioned from the benchmarks used in [1]. Those benchmarks were originally created using Xilinx's System Generator. For the experiments here, the Subsystem construct found in System Generator was used and whole subdesigns from the original benchmarks were converted to large hard macros. This had the benefit of guaranteeing that the resulting circuits (small macro-based and large macro-based) were logically identical so direct comparisons could be made, with any differences attributable to the difference in macro size. A few changes were made to HMFlow to facilitate the generation of larger hard macros, mainly the processing of complete subsystems as opposed to HMFlow's original behavior of flattening the design hierarchy. We call this new HMFlow variant "HMFlow-Large" to distinguish it from the original small-macro version of HMFlow of [1] which we will call "HMFlow-Small" hereafter.

Of the six benchmark designs selected for demonstrating HMFlow-Large, three were logically equivalent to three of the original benchmarks used to demonstrate HMFlow-Small [1], namely frequency_estimator (f_est), multi-band_correlator (mul_cor) and trellis_decoder (trel_dec). The other three benchmarks, brik1, brik2 and brik3 each represent an entire design that occupied an entire FPGA in the original telemetry receiver [10] selected as the source for the benchmark circuits.

Large hard macros were created in each benchmark by encapsulating major components into subsystems, often while

TABLE I: Slice Counts for Large Hard Macro Benchmarks

Design Name	Virtex 5 Slices
f_est	3970
trel_dec	5929
brik3	7505
brik2	7552
mul_cor	8258
brik1	9598

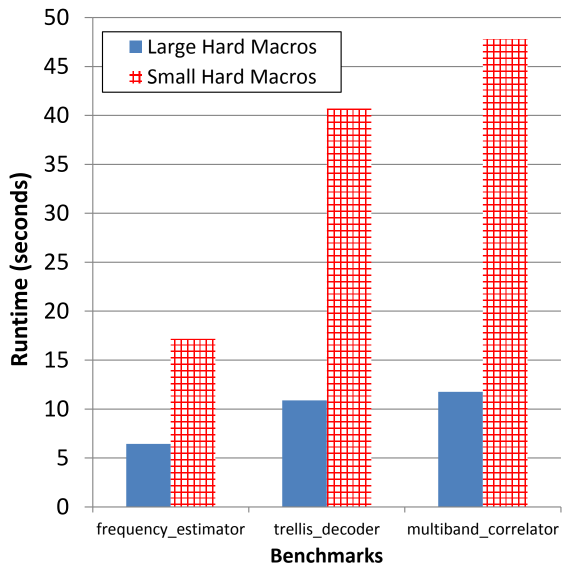

Fig. 5: Run-Time Comparisons of Three Benchmarks: Large Versus Small Hard Macros

also inserting pipeline registers at their boundaries. After all the subsystems were finalized, all six benchmarks were built using HMFlow-Large targeting a Virtex 5 SX240T. The slice count of the six benchmark designs is shown in Table I.

A. Comparisons of HMFlow-Small vs. HMFlow-Large

CAD tool run-times for the benchmarks and their maximum clock rates are graphed in Figures 5 and 6. As can be seen in Figure 5, using larger hard macros over smaller hard macros can reduce compilation times significantly, by at least 2.5–4×. This is a very significant reduction in that the smaller hard macro versions already compile at least 10× faster than the fastest Xilinx compilations. This validates our conjectures that large hard macros can speed up the implementation process, by as much as 30-40× for these benchmarks.

Figure 6 shows that using larger hard macros improves the maximum achievable clock rate by 60–70%[1]. We attribute this to the fact that more of the designs' routes are being

[1]It is possible that a small amount of this improvement (10%-15%) may have been due to the insertion of additional pipeline stages. However, these benchmarks had already been heavily optimized and it is far more likely that the majority of this improvement was due to the improved placement of internal circuitry in the larger hard macro by the Xilinx software.

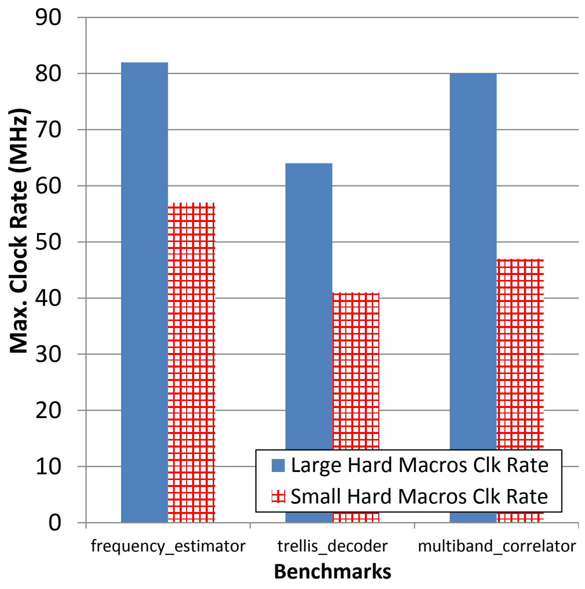

Fig. 6: Clock Rate Comparisons for Three Benchmarks: Large Versus Small Hard Macros

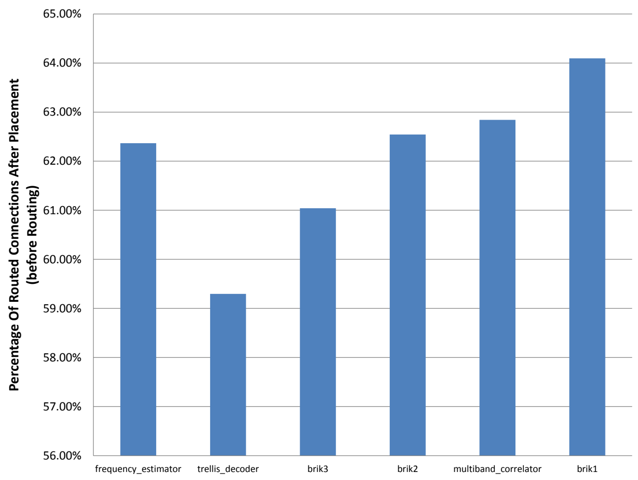

Fig. 7: The Number of Existing Routed Connections in the Large Hard Macro Benchmarks as a Percentage to Total Connections

packed into the hard macros by the higher quality Xilinx router as opposed to the lower quality HMFlow router which is used for making the macro-to-macro connections.

Some of the benefit attributed to the larger hard macros is the increased number of routed nets encapsulated within the hard macros. The large hard macro benchmarks include over 3.7× the routed connections as found on average in the smaller hard macro benchmarks. This removes a significant burden from off the HMFlow-Large router as the large hard macro benchmarks have over 60% of their connections already routed (not including clock, power and ground nets) before arriving at the HMFlow routing stage. The percentage of routed connections in each benchmark is shown in Figure 7.

B. Large Hard Macro Compilation (Creation) Time

Large hard macros generally take longer to create initially than small hard macros. Some compile times for larger hard macros are found in Table II.

TABLE II: Coarse-grained Hard Macro Compile Times

Hard Macro Name	Tiles Occupied	Compile Time
digital_gain_amplifier	62	160s
new_correlator_fifo	110	168s
ad_dynamic_range_calc	34	177s
Polyphase_filter	610	200s
pd_control	117	230s
Reindexer	645	260s
Aliasing_DDC	491	320s
multi_band_correlator1	12672	1324s

Although large hard macros have longer compile times than small hard macros, they are able to capture routing configurations which meet difficult timing constraints and which may have taken the Xilinx router hours to produce. Each time the hard macro is reused, that timing closure process does not need to be repeated, especially as bugs are found and fixed in *other* parts of the design.

C. Comparisons of HMFlow-Large vs. Xilinx

With large hard macros making a positive impact in performance and quality of HMFlow implementations, it raises the question, how close is this combination to the implementations produced by Xilinx? To answer this question, the remaining three benchmark designs (brik1, brik2 and brik3) were also compiled using HMFlow-Large and then all six benchmarks were also compiled with the Xilinx tools. The average speedups for HMFlow-Large and large hard macros over the Xilinx tools are shown in Table III and is about 52×. This is substantially more speedup than what was achieved using the smaller hard macros in [1].

Clock rates for HMFlow-Large circuits are 60–70% faster than those of HMFlow-Small due to the use of larger hard macros. This supports our conjecture that larger hard macros should lead to higher performance circuits. That said, the clock rates for our HMFlow-Large circuits still significantly lag those of the Xilinx-produced circuits as shown in Table IV.

TABLE III: Runtime Comparison of HMFlow-Large vs. Xilinx

	f_est	trel_dec	brik3	brik2	mul_cor.	brik1
Xilinx	392s	461s	605s	852s	498s	849s
HMFlow	7.0s	10.9s	12.9s	13.1s	11.8s	14.2s
Speedup	56×	42×	47×	65×	42×	60×

TABLE IV: Clock Rate Comparison of HMFlow-Large vs. Xilinx (MHz)

	f_est	trel_dec	brik3	brik2	mul_cor.	brik1
Xilinx	237	225	225	199	243	207
HMFlow	82	64	64	80	80	81
Slowdown	2.9×	3.5×	3.5×	2.5×	3.0×	2.5×

V. Conclusions and Future Work

Until now, our goal for HMFlow has been to develop the fastest prototyping flow possible without too much regard to quality of results (as measured by circuit clock rate). But, with these results we now have two data points showing what is possible. For the small macros of [1] HMFlow run-times are about 10× less than Xilinx. For the large hard macros of this work, HMFlow run-times are about 50× less than Xilinx with clock rates between 2.5-3.5× as slow.

Our ongoing and future work will focus on further exploring the tradeoff between tool runtime and circuit quality with the goal of providing an understanding of the range of trade-offs possible. This, in turn, holds promise for understanding how different design flows can be employed for different FPGA use models such as rapid prototyping and debug, high-performance computing, ASIC replacement, etc.

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*, pp. 117 –124, May 2011.

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

[3] E. L. Horta and J. W. Lockwood, "Automated Method to Generate Bitstream Intellectual Property Cores for Virtex FPGAs," in *Proc. Field Programmable Logic.2004*, 2004.

[4] Y. E. Krasteva, F. Criado, E. d. l. Torre, and T. Riesgo, "A Fast Emulation-Based NoC Prototyping Framework," in *RECONFIG '08: Proceedings of the 2008 International Conference on Reconfigurable Computing and FPGAs*, (Washington, DC, USA), pp. 211–216, IEEE Computer Society, 2008.

[5] 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*, pp. 119–124, September 2008.

[6] J. Coole and G. Stitt, "Intermediate Fabrics: Virtual Architectures for Circuit Portability and Fast Placement and Routing," in *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/software Codesign and System Synthesis*, CODES/ISSS '10, (New York, NY, USA), pp. 13–22, ACM, 2010.

[7] R. Tessier, "Fast Placement Approaches for FPGAs," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 7, no. 2, pp. 284–305, 2002.

[8] 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*, pp. 151–156, Aug.

[9] J. T. Lamprecht, "FPGA floor-planning impact on implementation results," Master's thesis, Brigham Young University, 2012.

[10] C. Lavin, B. Nelson, J. Palmer, and M. Rice, "An FPGA-based Space-time Coded Telemetry Receiver," in *Aerospace and Electronics Conference, 2008. NAECON 2008. IEEE National*, pp. 250–256, July 2008.

[11] C. M. Lavin, *Using Hard Macros to Accelerate FPGA Compilation for Xilinx FPGAs.* PhD thesis, Brigham Young University, 2012.