Improving Clock-Rate of Hard-Macro Designs

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*—HMFlow reuses precompiled circuit modules (hard macros) and other techniques to rapidly compile large designs in a few seconds - many times faster than standard Xilinx flows. However, the clock rates of designs rapidly compiled by HMFlow are often significantly lower than those compiled by the Xilinx flow. To improve clock rates, HMFlow algorithms were modified as follows: (1) the router was modified to take advantage of longer routing wires in the FPGA devices, (2) the original greedy placer was replaced with an annealing-based placer, and (3) certain registers were removed from the hard-macro and moved into the fabric to reduce critical-path delays. Benchmark circuits compiled with these modifications can achieve clock rates that are about 75% as fast as those achieved by Xilinx, on average. Fast run-times are also preserved; the improved algorithms only increase HMFlow run-times by about 50% across the benchmark suite so that HMFlow remains more than 30× faster than the standard Xilinx flow for the benchmarks tested in this paper.**

I. Introduction

FPGA devices are able to implement designs which often run orders of magnitude faster than software implementations of the same algorithms and thus provide an attractive alternative for a variety of computations. However, when FPGA development flows are compared to software development flows, the most glaring shortcoming FPGA's exhibit is their long development times. One reason for the long development times for FPGA designs is due simply to the very long compilation times required — sometimes requiring many hours to compile a single design iteration. Needless to say, such long compilation times are incompatible with interactive development flows such as are found in the world of software programming.

Commercial FPGA implementation tools typically flatten the entire design and reuse little, if anything, from previous runs. Further, the notion of precompiled libraries of predefined FPGA functions is largely absent in FPGA development flows. The net result however, is that commercial FPGA tools are able to produce very high quality designs (as measured by the density and clock rate of the final circuit), the major downside being the long compilation times required.

One approach to address FPGA compilation times is to use a building block approach — to construct an FPGA design as much as possible out of pre-built building blocks called *hard macros*. Hard macros are fully placed and routed circuit modules containing many logic elements and wires, but which can be placed as a unit onto the FPGA fabric. The motivation for this approach is that placing and routing a relatively small number of pre-built hard macros should be much faster than flattening and then placing and routing 10,000's of circuit elements (as the current commercial approaches do). An interesting research question with this focuses on the tradeoff between compilation time and circuit quality achievable with such an approach.

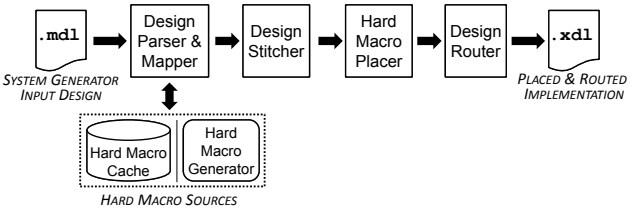


Fig. 1: HMFlow Block Diagram

The HMFlow system [1] uses such a hard macro-based approach to rapidly assemble FPGA designs. A block diagram of HMFlow is shown in Figure 1. Design entry is accomplished using the Xilinx SystemGenerator tool. HMFlow then parses the SystemGenerator Simulink design representation and, for each building block required either finds a corresponding hard macro in its cache or creates one. Ideally, all needed hard macros are found in the cache and the tool need only place and route the macros together to form a finished circuit. When needed, however, HMFlow calls on the Xilinx tools to generate new hard macro building blocks and places them into its cache for later use.

The goal of HMFlow, to date, has been to create the fastest synthesis/place/route tool chain possible without regard to the clock rate of the compiled circuit. The earliest version of HMFlow demonstrated speedups of over 10x compared to the Xilinx flow for synthesis/placement/routing but produced circuits that ran at only a fraction of the clock rate. In that work, the hard macros were relatively fine grained and consisted of blocks such as adders, muxes, registers, comparators, etc. Later work [2] investigated the use of much coarser hard macros (FIR filters, FFT blocks, ...) and demonstrated speedups of 40-60× over the Xilinx tool flow while producing designs that were 2.5x to 3.5x slower than achieved by Xilinx.

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.

This paper reports on improvements to the HMFlow algorithms that improve clock rates of the finished design while still preserving reduced compilation times. Specifically, this paper demonstrates that benchmark circuits compiled by HMFlow with these improved algorithms can achieve clock rates that are about 75% of those achieved by the standard Xilinx flow, on average. Fast run-times are also preserved; the improved algorithms only increase HMFlow run-times by about 50% across the benchmark suite so that HMFlow remains more than 30× faster than the standard Xilinx flow for the benchmarks tested in this paper.

Three major changes were made to HMFlow to improve clock rates: (1) the router was modified to take advantage of longer routing wires in the FPGA devices, (2) the original greedy placer was replaced with an annealing-based placer, and (3) where advantageous, registers were removed from the hard-macro and moved into the fabric (a form of retiming) to reduce the delay for some critical paths. The balance of this paper is as follows: after first reviewing related work, the next three major sections of the paper (Placer Improvements, Router Improvements, Register Replacement) discuss these modifications in detail. The Analysis Section then analyzes the impact of these modifications both separately and in combination.

II. 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 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. Placer Improvements

The original HMFlow placer was based upon greedy heuristics and was designed to achieve minimal run-time but still achieve reasonable results. Unlike conventional FPGA placers, the HMFlow placer deals with many fewer placeable objects (perhaps only 30 - 40 objects) and it was assumed that suitable placement could be achieved with much less computational effort than that required for a conventional FPGA placer. Previously-reported results validated this assumption though lowered placement quality generated by greedy heuristics was likely one of the primary factors that limited the clock frequencies initially achieved by HMFlow to be 2.5 to 3.5× less than that achieved by the longer Xilinx compilations.

To determine how much quality was being lost due to the simple greedy algorithms used in the original HMFlow placer, a much slower annealing-based placer was implemented in this work and compared against the earlier greedy placer. The annealing-based placer achieved much higher clock frequencies at a cost of much longer run-times (minutes versus seconds) and demonstrated that placement of hard macros could be further improved to increase clock rates. Through experimentation, it was determined that accelerating the schedule of the simulated-annealing placer so it finished much sooner had very little impact on the overall

quality of the resulting implementation. This, combined with a bounding box optimization (to detect overlap) for the hard macros created a modified simulated annealing placer which produced significantly better results compared to the earlier HMFlow placer while requiring only a modest increase in runtime. That placer is the subject of the remainder of this section.

A. Basic Algorithm Details

Simulated annealing begins by creating some initial placement, S, as a starting point for the algorithm (see Algorithm 1). A starting temperature, t, is chosen or calculated and t is generally a large value that allows the algorithm to explore many inferior solutions during the early stages of the search. The algorithm then repetitively repeats the following steps:

1) The placement is modified, usually by swapping placements between primitives.
2) After each modification, the current placement is evaluated by computing some global statistic, e.g., total wirelength, that serves as a proxy for overall placement quality.
3) If the move improves placement, it is accepted, otherwise it is accepted with some probability that is a function of the current temperature.
4) a new temperature is computed according to the cooling schedule.

After $movesPerTemperatureStep$ moves has been made, the temperature is decreased by some percentage, $temperatureReduceRate$. This process repeats over and over until some stopping criteria is met which is often when the system cost does not change after several moves (see Algorithm 1).

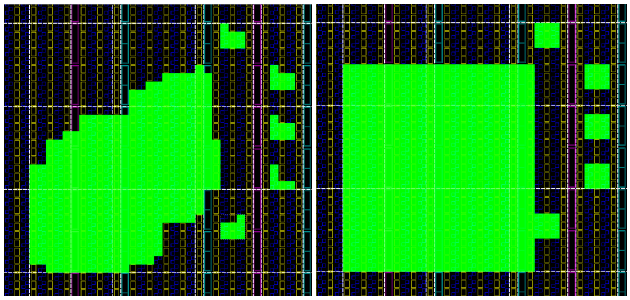
1: $S \leftarrow createInitialPlacedSolution()$
2: $t \leftarrow initialStartingTemp$
3: **while not** $reachedStoppingCriteria()$ **do**
4: **for** $i = 0$ to $movesPerTemperatureStep$ **do**
5: $S_i \leftarrow generateRandomMove()$
6: **if** $Cost(S) \geq Cost(S_i)$ **then**
7: $S \leftarrow S_i$
8: **else if** $e^{(Cost(S)-Cost(S_i))/t} \geq random[0,1)$ **then**
9: $S \leftarrow S_i$
10: **end if**
11: **end for**
12: $t \leftarrow t * temperatureReduceRate$
13: **end while**
14: **return** S

Algorithm 1: Basic Simulated Annealing-Based Placement

B. Customization of Simulated Annealing for Hard Macros

The algorithm for the simulated annealing placer for HMFlow and its hard macros deviated from the basic algorithm shown in Algorithm 1. The customizations are listed below:

1) The number of moves per temperature step varies based on the acceptance rate of the moves being generated. The number of moves per temperature step increases when the acceptance rate is near a more productive acceptance rate which was determined to be approximately 44% [8].
2) The inner **for** loop was replaced with a **while** loop as the moves per temperature step were only counted when they were accepted moves rather than counting total moves.
3) The initial starting temperature was set to a value equal to 1.5 times the initial system cost.
4) The process stopped if either the move acceptance rate fell below 2%, or, the temperature had dropped below a value of 0.01.
5) The cost function used was the sum total measure of Manhattan distances between all port connections in between hard macros.



(a) (b)

Fig. 2: (a) Representation of a Set of Hard Macros Drawn with a Tight Bounding Box (b) An Approximated Bounding Box for the Same Hard Macros for Accelerating the Simulated Annealing Hard Macro Placer

C. Challenges of Hard Macros in a Simulated Annealing Placer

Computing the legal placement of hard macros is one of the challenges of placing hard macros. Typical non-hard-macro FPGA designs are composed primarily of primitive instances. Instances are placed on compatible primitive sites and the check required to see if the placement is valid simply requires a comparison to make sure the primitive site is compatible and that it is unoccupied. However, when a hard macro is moved, it may contain dozens or even hundreds of primitive instances in addition to many routed nets. In a potential move, each instance must be checked for a compatible site which is unoccupied, and each PIP within each routed net must also be verified to make sure the move is valid.

Two techniques were used to reduce HMFlow placer runtime. First, during the creation of each hard macro, all of its valid placement locations were pre-computed off-line and stored with the hard macro in the hard macro cache. This saved runtime when initializing the placer and allowed the move-generator to choose from a set of a valid placement locations rather than randomly choosing locations that often would not support the hard macro.

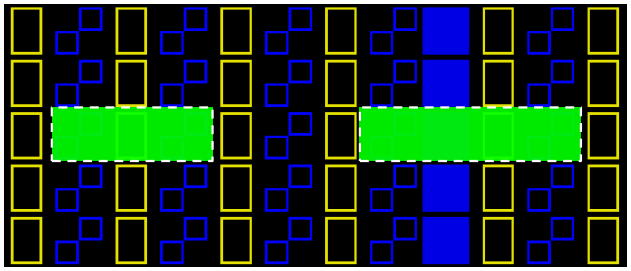


Fig. 3: An Illustration of the Problem of an Approximating Bounding Box Where the Box Changes Size Based on Location

Second, rather than using a tight bounding box around each hard macro as shown in Figure 2a, each hard macro had an approximating bounding box calculated around all of its logic and routing as shown in Figure 2b. The bounding boxes accelerated move validation in that only the bounding box needed to be moved and tested for overlap with other bounding boxes in the design rather than each primitive instance and PIP in the hard macro. The bounding box approximation did restrict to some degree the flexibility of how hard macros could be placed—the bounding boxes contained some empty space making for some inefficiencies—but the trade-off reduced runtime to make it a feasible placement technique.

Another challenge that arose from using the approximating bounding box was that the Xilinx FPGA fabric is not always uniform and can skew the bounding box sizes based on where it was originally calculated. To illustrate this concept, consider Figure 3. On the left of the figure is the outline of a hard macro that consumes two horizontally adjacent CLB tiles with a switchbox tile between them. This would create a bounding box of 1 tile high and 3 tiles wide. However, now consider the same hard macro placed at a different location as shown on the right side of Figure 3. An extra column of configuration tiles is found in between the two columns of CLBs. This creates a bounding box of 1 tile high and 4 tiles wide.

The problem occurs when a hard macro with the smaller 1x3 bounding box is calculated and then moved to a spot where it should have a 1x4 bounding box. When hard macros are big enough, they can span multiple columns and rows of tiles that ultimately cause the bounding box to be too small for a specific placement. This problem, on rare occasion, can lead to overlap of hard-macro placements which results in placement failure.

In trying to remedy this issue, it was determined that compensating for the placement-varying bounding box would eliminate much of the runtime savings it created, thus an alternative solution was chosen. The solution implemented was that all hard macros are checked for valid placements at the end of the simulated annealing process. If any hard macros are part of an invalid placement (overlap) the smaller of the two hard macros is replaced at the closest valid location to its final placer-decided location. The situation occurs quite infrequently (once or twice in approximately 10 placement runs) and does not significantly impact placement quality but does allow the preservation of the bounding box runtime acceleration optimization.

IV. Router Improvements

One of the major inefficiencies found in the original HMFlow router was the fact that it often did not use long line routing resources efficiently when connections had to be made over a long distance. This resulted in long distance connections being routed with several shorter hops of less efficient resources that ultimately added up to a significant amount of propagation delay for the net. This behavior was mostly a side effect of the driving force behind the router's original implementation which was "route as quickly as possible."

1) Long Lines: A long line in Xilinx FPGAs is the longest routing wire available. In the Virtex 4 architecture, long lines spanned 24 switch boxes, however, they were reduced in length to 18 switch boxes in the Virtex 5 architecture. They are available in both the horizontal and vertical directions. A long line is also capable of connecting to every 6th switch box in its path as illustrated in Figure 4.

By using long lines, routing connections that are very far apart can be connected with relatively few long lines. In Virtex 4, the next longest wire is the hex line providing a connection distance of 6 switch boxes. In Virtex 5, the next longest routing resource is the pent line which provides a connection distance of 5 switch boxes.

Unfortunately, timing information for the wires found in Xilinx FPGAs is not available publicly and therefore, wire delay cannot be used for evaluating the quality of different routes. However, it is clear that the number of wire segments used in routing a connection can often have a bigger impact on delay rather than a connection's length. Thus if the use of a long line can significantly reduce the number of total wire segments in the routed connection, it will also significantly reduce delay.

2) Long Line Router: To overcome the long distance routing inefficiency of the router, a specialized long line router was developed to specifically route long distance connections during the routing process. This long line router

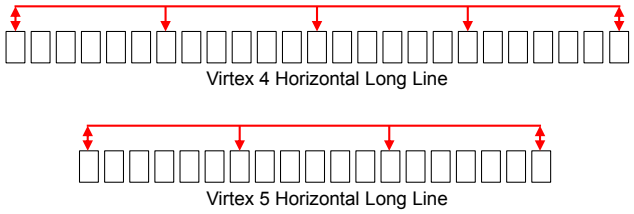


Fig. 4: Representation of Virtex 4 and Virtex 5 Long Line Routing Resources

finds very good routes using as many long lines as possible to get close to the sink of the connection. Once the route has used as many long lines as possible, the routing is then finished by the main routing algorithm.

The long line router is essentially a maze router that only uses long line resources. It is invoked by the main router when it encounters a connection to be routed that has a Manhattan distance of 12 switch boxes or more between source and sink. Through preliminary testing on a handful of designs and routes, it was determined that the minimum distance for which the long line router provided benefit was to invoke it for distances of 12 switch boxes or more. Long lines can only be used in discrete hops of 6, 12 and 18 hops as previously shown in Figure 4 and if a route is only using 6 hops worth of routing, it still incurs the delay of 18 hops. Therefore, setting a threshold of 12 would reduce the likelihood of the long line router utilizing a long line for a distance of only 6 hops. The goal of the threshold was simply to invoke the long line router when it would be likely to provide benefit and avoid situations where it could introduce unnecessary delay into a route.

Once the long line router is invoked, the first task is to find the closest and most efficient entry point to a nearby long line. If no available long line resource is available, the long line router fails and returns the routing task to the main router to complete the net without long line optimization. This occurs quite infrequently, but it does happen when congestion is high around the source of the connection.

Once an efficient path to a long line is found, the long line router will change modes to only search out a path using long line resources. Again, the algorithm is based on a maze router and will end prematurely if too much congestion is encountered. However, if only a partial route using long lines is found, the long line router will still provide the partial route to the main router to use as its starting point.

When the long line router exhausts the options to find the closest long line exit to the sink, it returns the partially routed path back to the main router to finish the final connection. Regardless of whether the long line router succeeds in finding a partial or complete long line path, the main router's parameters change to try and obtain a higher quality path as the longest paths are more likely to become a critical path in the final implementation. These changes do increase router runtime by about 15-20% on average.

V. Register Replacement

One of the biggest problems of large hard macro placement is the inherent presence of long distance connections between hard macros that often become critical paths in design implementation. The placer can reduce these paths to some extent, however, because of the rigid nature of hard macros, the placement of individual logic elements (and therefore net connection points) is not nearly as fluid and malleable as in conventional design compilation flows.

To try to reduce the effects of this problem a form of retiming, called register re-placement, is used. Register re-placement occurs after the placement phase but before routing in HMFlow and consists of moving registers at the boundaries of hard macros out into the FPGA fabric to break up long wires, reducing their delay. The basic process for register re-placement is given in Algorithm 2.

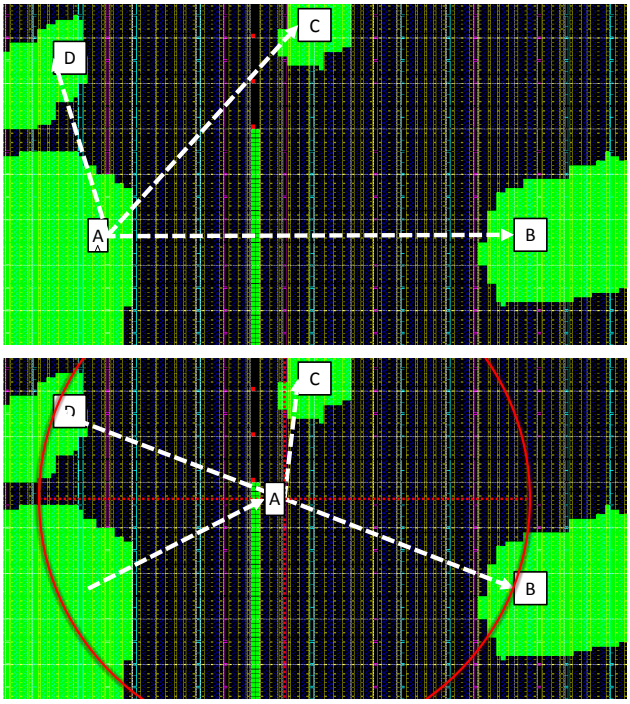


Fig. 5: A Simple Example of a Register (A) Being Re-placed at the Centroid of the Original Site of Register A and Sinks B, C and D

1: $C \leftarrow identifyHardMacroPortConnections()$
2: $T \leftarrow 10$
3: $S \leftarrow \{\}$
4: **for each** $Connection\ c\ in\ C$ **do**
5: **if** $c.getTotalLength() \geq T$ **then**
6: $r_s \leftarrow c.getSourceRegister()$
7: $r_d \leftarrow c.getDestinationRegister()$
8: **if** $r_s \neq null$ **and not** $r_s \subseteq S$ **then**
9: $P \leftarrow getAllPinLocations(c)$
10: $MoveRegister(r_s, findCentroid(P))$
11: $S.add(r_s)$
12: **else if** $r_d \neq null$ **and not** $r_d \subseteq S$ **then**
13: $P \leftarrow getAllPinLocations(c)$
14: $MoveRegister(r_d, findCentroid(P))$
15: $S.add(r_d)$
16: **end if**
17: **end if**
18: **end for**

Algorithm 2: Optimal Register Re-placement

The process begins by identifying all external hard macro nets (those nets connected exclusively to hard macro ports).

From the nets, all connections (a connection being a single source pin to a single sink pin) can be extracted and then for all connections that are longer than a certain length (Manhattan distance from source pin to sink pin) of 10 tiles, register re-placement is considered for evaluation. If a connection has a register at its source pin, the register is moved to the centroid of all the net's source and sink pins.

To illustrate the centroid of a net and how it is used to move a register, consider Figure 5 where the top picture represents a portion of the FPGA fabric after a hard macro design has been placed. The output of register A in the large hard macro on the left, is driving three separate sinks B, C and D in three other hard macros. Due to the placement, the connections are very far apart, especially from register A to sink B. In the bottom picture of Figure 5 it shows register A moved to the centroid (the center of the smallest circle that still includes the 4 points). This re-placement of the register significantly decreases the longest path the router would have to route in this particular case, ultimately paving the way for a higher quality implementation.

VI. Analysis

Once all three of the HMFlow improvements were complete, it was necessary to analyze their impact on quality of result. The ultimate goal of these improvements was to improve implementation clock rate while limiting CAD tool runtime increase. In order to accurately and fairly measure quality of result of each improvement, a variety of configurations were created and run with all 6 of the large hard macro benchmark designs from [2].

A. Methods Used to Compare Results

Because of the random variation implicit in simulated annealing, each benchmark configuration is evaluated and compared using three different methods. First, each configuration is compared using the results from a single default seed for both Xilinx and HMFlow. Second, results are compared using the arithmetic mean from 100 distinct runs with different seeds used for each run. Finally, the results are compared using the implementations that achieved the highest clock rate from the previous 100 distinct runs of place/route for both Xilinx and HMFlow.

One of the major reasons for providing all three metrics is that some designers may have a cluster of computers suitable for farming out the 100 jobs to run in parallel. In this scenario, the best of 100 results could be obtained rather quickly and would be the metric of interest. However, other designers may not have such a luxury and would be more interested in the average (or expected) result. To aid decision making to both groups of designers, both metrics are included in addition to the single run result when relevant.

B. Results of Three HMFlow Improvements

To get a sense of the impact of each improvement added to HMFlow, all 8 possible configurations of the three improvements within the flow were tested separately on all 6 benchmarks. The first configuration (Configuration 0) is the absence of all three improvements. This configuration is considered the baseline against which all 7 other configurations are compared. All of the configurations are summarized in Table I.

TABLE I: HMFlow Improvement Configurations

Config. Name	Placer		Register Replace	Router	
	Orig.	Annealer		Orig.	Long Line
C0 (baseline)	X			X	
C1		X		X	
C2	X		X	X	
C3	X				X
C4		X	X	X	
C5	X		X		X
C6		X			X
C7 (Best)		X	X		X

As can be seen from Table I, there are two placers (the original heuristic hard macro placer and the new simulated annealing hard macro placer introduced in this paper), an optional register re-placement step, and two routers (the original router and the new long line-enabled router discussed in this paper). All 8 configurations were tested with the 6 large hard macro benchmark circuits. Those configurations that did not use the simulated annealing hard macro placer (C0, C2, C3, and C5) were not run 100 times as the heuristic placer was not dependent on a random seed as its input.

C. Results of Optimizing Improvements

The results of the eight configurations of HMFlow improvements are shown in Tables II, III, and IV. Each table includes results from the Xilinx tools compiling the benchmarks to provide a comparison. The Xilinx tools were run in 'Performance Evaluation Mode' that attempts to obtain good results in a reasonable amount of time without a timing constraint. This mode was chosen because it most closely matched the goals of HMFlow.

Table II represents clock rates obtained for each of the benchmarks when executing a single default compilation run of the tools. The six benchmarks used were taken from a large MIMO communications design, and range in size from 4,000 to 9,600 Virtex-5 slices. See [2] for more details.

Table III shows the average clock rates obtained by compiling each benchmark 100 times, each compilation using a different seed or table cost entry. For those configurations where a seed does not have an effect (heuristic placer configurations, C0, C2, C3, and C5), the values are the same as those in Table II. Interestingly, the single run and average of 100 run results are very similar in their average clock rates and average improvement in clock rates over the baseline, C0. This observation holds true also for the Xilinx tool's results.

Table IV represents the clock rates obtained from the best of 100 compilation runs. These results could be obtained

very quickly if all 100 compilation runs could be run in parallel such as on a supercomputer or cluster. When compared with averages of the single and average of 100 runs, the best of 100 runs results are approximately 50% better. This provides a significant advantage to those with easy access to parallel computing power. However, the advantage is less if the designer is using the Xilinx tools which only produce approximately 25–30% better clock rates.

D. Impact of Improvements

The simulated annealing hard macro placer had the single greatest impact on performance of the three improvements described. Compared to baseline (C0) implementations, replacing the heuristic placer with the simulated annealing version (C1) increased clock rates on average by 50%. The next most impactful improvement was the addition of the register re-placement step (C2) that improved clock rates by 43%. The improvement of the long line optimized router (C3) had the smallest impact when measured in isolation, providing improved clock rates of only 22%. These results illustrate the impact a good or bad placement can have on the overall implementation clock rate. Ultimately, however, the HMFlow router is limited by the lack of detailed device timing information.

Dual combinations of the improvements (C4, C5, and C6) were closely additive in their improvement of clock rate showing how each improvement approach was mostly independent of the others. C7, the configuration which combined all three of the improvement techniques, gave the best clock rate improvement of $2\times$ or $2.5\times$ in the best of 100 runs case.

E. Runtime Results

Table V shows the runtime for each of the 6 benchmarks as compiled with the major HMFlow revisions presented in this chapter. Runtime is measured in seconds and is defined as the total time to compile a design starting from the time taken to read in a design's source files and ending after creating a placed and routed implementation file (XDL or NCD respectively).

As can be seen from Table V, the baseline C0 has the fastest runtime performance of any tool configuration. This is largely due to its usage of the lower quality but faster placer and router algorithms present in the earlier unoptimized version of HMFlow. When the newer simulated annealing, register re-placement and long-line optimized router are introduced (C7), runtimes increase by a little less than 50%.

Further analysis (not shown due to length restrictions) showed that the variability of quality of results achieved with HMFlow was much higher than that achieved by the Xilinx tools. Though this variability is not a concern when selecting the best compilation from 100 runs of HMFlow, it is a problem if the designer only wants to run HMFlow once. The T3 variation of HMFlow was created in response to this.

TABLE V: HMFlow Run-Time vs. Xilinx Run-Time

Benchmark	C0	C7	T3	Xilinx
frequency_estimator	6.82s	10.01s	10.91s	392.06s
trellis_decoder	11.81s	18.2s	18.57s	461.34s
brik3	13.81s	28.56s	27.81s	605.16s
brik2	15.26s	20.94s	19.98s	851.61s
multiband_correlator	12.45s	18.82s	18.8s	497.91s
brik1	15.98s	15.56s	15.66s	848.79s
Average Runtime	12.69s	18.68s	18.62s	609.48s
Speedup (over Xilinx)	48.0×	32.6×	32.7×	-

TABLE VI: Clock Rate Summary: HMFlow versus Xilinx

Benchmark	C0	C7	HMFlow(T3)	Xilinx
frequency_estimator	116	194	182	227
trellis_decoder	62	128	166	251
brik3	66	122	131	241
brik2	82	153	173	203
multiband_correlator	67	146	196	250
brik1	65	189	199	200
Average Clock Rate (MHz)	76	155	175	229

T3 is essentially the same as the C7 version of HMFlow but has an additional term in the placer cost function. This additional cost function term takes into account the length of the single longest wire in the design, in addition to total wire length (as in the C7 cost function). This addition did not appreciably affect the runtime of T3 compared to C7 as shown in Table V. However, it did significantly improve the quality of result as shown in Table VI, which is an abbreviated version of Table II but also including T3 results. T3 (175 MHz) did not achieve as high a clock rate as the best of 100 runs (196 MHz, from Table IV), but did much better for a single run (155 MHz, from Table II) .

Overall, the results can be viewed as a positive outcome for HMFlow. The final configuration of HMFlow (T3) can produce implementations over $30\times$ faster than the Xilinx tools and still obtain clock rates that are 75% of the implementations produced by the best Xilinx efforts. Put another way, HMFlow produces a placed and routed implementation that runs 175 MHz on average and can be obtained in less than 20 seconds. This is in contrast to Xilinx which will take over 10 minutes to produce a design that will only run about 30% faster. This is a good result for HMFlow as it demonstrates that rapid compilation is very feasible for FPGA CAD.

VII. Conclusions

HMFlow with the new optimized versions of the router, placer and the addition of register-replacement, improved average clock rates by almost $3\times$ over the earlier version of HMFlow. This was accomplished while delivering compilation times that are still over $30\times$ faster than the conventional Xilinx flow. Given these performance numbers, HMFlow offers an attractive alternative to conventional FPGA compilation techniques and has the potential to increase designer productivity with its rapid compilation benefits.

TABLE II: HMFlow Benchmark Clock Rates of Single (Default) Run (in MHz)

Benchmark	C0	C1	C2	C3	C4	C5	C6	C7	Xilinx
frequency_est.	116	146	156	128	189	165	177	194	227
trellis_decoder	62	88	88	74	111	114	123	128	251
brik3	66	118	77	84	115	81	132	122	241
brik2	82	99	122	101	139	141	113	153	203
multiband_corr.	67	83	87	94	127	109	114	146	250
brik1	65	153	127	80	167	144	171	189	200
Average	76	115	109	93	141	126	138	155	229
Improvement	-	1.5×	1.43×	1.22×	1.85×	1.65×	1.81×	2.04×	2.99×

TABLE III: HMFlow Benchmark Clock Rates of Average of 100 Runs (in MHz)

Benchmark	C0	C1	C2	C3	C4	C5	C6	C7	Xilinx
frequency_est.	116	144	156	128	170	165	167	186	220
trellis_decoder	62	100	88	74	124	114	123	139	247
brik3	66	108	77	84	116	81	119	123	238
brik2	82	91	122	101	131	141	115	153	198
multiband_corr.	67	101	87	94	151	109	124	177	253
brik1	65	144	127	80	169	144	169	168	203
Average	76	115	109	93	143	126	136	158	226
Improvement	-	1.5×	1.43×	1.22×	1.88×	1.65×	1.78×	2.06×	2.97×

TABLE IV: HMFlow Benchmark Clock Rates of Best of 100 Runs (in MHz)

Benchmark	C0	C1	C2	C3	C4	C5	C6	C7	Xilinx
frequency_est.	116	194	156	128	208	165	217	228	264
trellis_decoder	62	124	88	74	157	114	157	176	280
brik3	66	139	77	84	146	81	146	157	260
brik2	82	134	122	101	167	141	160	180	205
multiband_corr.	67	148	87	94	207	109	172	223	270
brik1	65	191	127	80	214	144	205	212	214
Average	76	155	109	93	183	126	176	196	249
Improvement	-	2.03×	1.43×	1.22×	2.4×	1.65×	2.31×	2.57×	3.26×

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, B. Nelson, and B. Hutchings, "The Impact of Hard Macro Size on FPGA Clock Rate and Place/Route Time," in *Field-Programmable Logic and Applications, 23rd international Conference on*, p. to appear, Sep 2013. Removed for blind review.

[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. Lam and D. Jean-Marc, "Performance of a new annealing schedule," in *Proceedings of the 25th ACM/IEEE Design Automation Conference*, DAC '88, (Los Alamitos, CA, USA), pp. 306–311, IEEE Computer Society Press, 1988.