HMFlow: Accelerating FPGA Compilation with Hard Macros for Rapid Prototyping

Christopher Lavin, Marc Padilla, Jaren Lamprecht, Philip Lundrigan,
Brent Nelson and Brad Hutchings
NSF Center for High-Performance Reconfigurable Computing (CHREC)
Dept. of Electrical and Computer Engineering
Brigham Young University
Provo, UT, 84602, USA
Email: {chrislavin, brent_nelson, brad_hutchings}@byu.edu

***Abstract*—The FPGA compilation process (synthesis, map, place, and route) is a time consuming task that severely limits designer productivity. Compilation time can be reduced by saving implementation data in the form of hard macros. Hard macros consist of previously synthesized, placed and routed circuits that enable rapid design assembly because of the native FPGA circuitry (primitives and nets) which they encapsulate.**

This work presents results from creating a new FPGA design flow based on hard macros called HMFlow. HMFlow has shown speedups of 10-50X over the fastest configuration of the Xilinx tools. Designed for rapid prototyping, HMFlow achieves these speedups by only utilizing up to 50 percent of the resources on an FPGA and produces implementations that run 2-4X slower than those produced by Xilinx. These speedups are obtained on a wide range of benchmark designs with some exceeding 18,000 slices on a Virtex 4 LX200.

I. Introduction

For years, hardware designers have looked on almost despairingly at the rapid compile times of their software engineering colleagues. While their software friends perform many compile-debug-edit cycles per day, they are lucky to get one per day, or sometimes, one per week. Faster implementation times would ultimately translate into improved productivity because hardware engineers would be able to test and debug more designs per day—just like their software counterparts. Unfortunately, FPGA implementation times are not getting much faster, largely because devices keep getting bigger with every generation.

One may argue, for verification purposes at least, that compilation time can be avoided simply by using simulation to verify correct function. Indeed, where possible, simulation can and should be the tool of choice. Compile times for simulators are similar in duration to conventional software compiles and simulators provide more convenient observability than FPGA devices. However, simulation executes RTL approximately 1,000,000 times more slowly than silicon. For complex designs, e.g., software radio, radar, print rendering, etc., it often takes too long to verify functional correctness in simulation. Engineers in these situations may initially perform RTL verification with a simulator, but later, as simulation time begins to be counted in days, these engineers start loading their designs into FPGAs so their algorithms can be tested in-system, with actual data.

One way to accelerate debug and verification time is to reduce the amount of computation required to convert RTL to a bitstream by using pre-compiled modules. For example, in software, pre-compiled modules take the form of large pre-compiled libraries that need only be linked to the final executable during the compile-debug-edit cycle. Pre-compiled modules are widely used in software compilation flows and they dramatically reduce build time by eliminating compilation, e.g., reducing the computational effort required to build a software application.

In hardware, pre-compiled modules are typically referred to as "hard macros." A hard macro is a module that has been pre-compiled (previously synthesized/mapped/placed/routed) and stored in a library for later use by a designer. Previous work [1] demonstrates the feasibility of a design methodology based on hard macros and exhibits the potential they have to accelerate FPGA compilation times for rapid prototyping purposes. This work further demonstrates the validity of this technique by implementing a complete FPGA design flow based on hard macros called HMFlow.

Our goal for HMFlow is to demonstrate a reduction in compilation time of at least 10× over the conventional Xilinx design flow. In order to focus on this goal, designs were optimized for fast compilation rather than clock rate and area. In addition, we only targeted designs which utilize 50% or less of FPGA resources for a given device. These tradeoffs are quite acceptable for rapid prototyping purposes where compilation time is a significant bottleneck.

The primary research questions answered in this paper are:

- What challenges exist in constructing a rapid FPGA compilation flow using hard macros?
- How can such challenges be overcome?
- How much faster is a hard macro-based compilation flow than a conventional flow?
- What impact does rapid compilation have on maximum clock rate and area?

This work was supported by the I/UCRC Program of the National Science Foundation under Grant No. 0801876.

The rest of the paper is outlined as follows: In Section II we describe similar efforts to accelerate FPGA compilation. Section III introduces the main challenge of creating a hard macro-based design flow for commercial FPGAs and a new open source FPGA CAD tool framework called RapidSmith. In Section IV we present and describe the new hard macro-based design flow called HMFlow, its challenges and how they were overcome to achieve rapid compilation. In Section V, the benchmark circuits and the performance of HMFlow on those circuits is presented and analyzed. In Section VI we conclude and describe future efforts of HMFlow.

II. Related Work

Accelerating place and route is one way to reduce overall compilation time. Three different approaches to reduce place and route time are: (1) use variants of sequential algorithms, (2) develop parallel versions of these algorithms, or (3) accept a lower Quality of Result (QOR). Among these approaches are techniques such as router parallelization [2], multiphase placement [3], accelerated simulated annealing [4][5], clustered hierarchical placement [6] [7], and accelerated routing [8]. In contrast to the others, [5], [6] and [7] have reported on results that trade QOR for reduced place and route time.

Another way to reduce FPGA implementation time is to simply reduce the amount of circuitry that needs to be placed and routed by using pre-compiled (previously placed/routed) circuits wherever possible. Horta and Lockwood [9] demonstrated the creation of bitstream-based re-locatable cores which are quite similar in nature to hard macros. Similar efforts are reported in [10] where bitstream hard cores were used in a network-on-chip to provide accelerated logic emulation and prototyping. Unfortunately, bitstream hard cores are much more restricted because they must reside between restrictive configuration boundaries and require matching bus macro interfaces to be present both in the core as well as in the existing FPGA configuration. Similar work by Tessier [6] shows usage of pre-placed macroblocks accelerate place and route by 2.6× over commercial tools. However, the macroblocks did not include any routing information.

Intermediate virtual fabrics are another strategy that exploits reuse of pre-compiled structures [11]. This technique abstracts the FPGA to a domain-specific fabric created off-line and customized for a particular application domain. 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, however, each newly created intermediate fabric still requires a lengthy place and route.

Our approach differs from those previously mentioned in that it uses non-traditional algorithms for placement and routing and preserves internal routing within hard macros to potentially reduce routing time. The tool described in this work is also compatible with System Generator, a commercially-available design capture system provided by Xilinx. By leveraging non-traditional algorithms, custom hard macros and System Generator, we demonstrate a new approach to accelerate FPGA compilation.

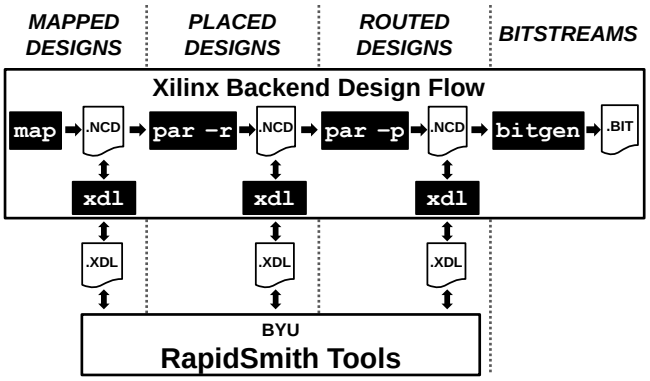

Fig. 1. Illustration of where XDL interfaces with the Xilinx tool flow and how RapidSmith interacts with XDL.

III. RapidSmith: An XDL Framework

Experimenting with actual Xilinx devices requires a new physical CAD flow (place/route) that supports hard macros. As reported by [1], the Xilinx placer does not handle hard macros well; when fed designs that included several hard macros, placement often failed, or exhibited excessively long runtimes. PlanAhead, the Xilinx physical floor-planning tool, is also not well-suited to our purposes because it does not automatically generate re-locatable hard macros. Xilinx does provide an exchange format, the Xilinx Design Language (XDL), which is an open design format and exposes enough device detail (however, without timing data) so that experimental place and route software can be written to directly target Xilinx devices. Although XDL is quite verbose, it is very useful and forms the basis of RapidSmith, a tool that can be used to rapidly prototype CAD tools for Xilinx devices.

Used to develop HMFlow, RapidSmith (1) imports designs in XDL, (2) provides an efficient API for manipulating XDL designs, e.g., sufficient to write a placer or router and (3) exports designs as XDL to the Xilinx software so a bitstream can be generated. XDL and RapidSmith played an important role in the development of HMFlow and are discussed in the remainder of this section.

A. Xilinx Design Language (XDL)

XDL is an interface provided by Xilinx which is a human-readable text-based language describing native circuits for Xilinx FPGAs. XDL describes circuit designs in the same fashion as the more widely used Netlist Circuit Description (NCD) file format. NCD and XDL files describe a native (physical) netlist for Xilinx FPGAs that contain mapped FPGA primitives and circuit networks (nets). These native netlists can represent designs which can exist at any phase of place or route. The XDL interface exists as a Xilinx executable distributed with its design tools called `xdl`.

The `xdl` program has three modes; one to convert NCD to XDL files (`-ncd2xdl`), one to convert XDL to NCD files (`-xdl2ncd`) and a `-report` mode. The first two modes

allow the user to gain access to designs in XDL and convert them back to NCD for use in the remainder of the Xilinx tool flow as can be seen from the top portion of Figure 1. The report mode produces very large (several gigabytes) of textual data in reports called XDLRC files which describe a particular FPGA device. Although XDLRC report files do not contain any timing information about a device, they do provide the user with enough information to create full, non-timing driven placement and routing tools.

Although XDL allows the user to access the proprietary NCD format and manipulate designs in XDL, there existed no framework at the time of our experimentation to easily build FPGA CAD tools[1]. The lack of support for XDL and its ungainly nature led to the creation of our own custom XDL tool framework called RapidSmith.

B. RapidSmith: A Framework for the Rapid Creation of FPGA CAD Tools

RapidSmith [14] is a Java-based software tool which leverages XDL and enables the rapid creation of FPGA CAD tools for Xilinx FPGAs. RapidSmith solves two major challenges of using XDL. First, RapidSmith provides a complete framework that can import XDL, modify designs and also export XDL to allow the design to be used in the remainder of the Xilinx tool flow. Second, it dramatically reduces the multi-gigabyte-sized XDLRC report files to a much more compact format suitable for the use and creation of custom FPGA CAD tools.

RapidSmith can import/export XDL designs anywhere in the Xilinx tool flow that accepts NCD as seen in Figure 1. RapidSmith includes a fast, custom-built XDL parser to import designs into its own data structure for manipulation. The RapidSmith design data structure has over 200 APIs specifically written for design manipulation. This design data structure is patterned heavily after XDL in that it uses the same design abstractions: primitive instances, nets, modules and module instances. The organization of these design abstractions in XDL and RapidSmith is shown in Figure 2. XDL designs (and thus, RapidSmith designs) are composed of a collection of primitive instances (such as SLICELs, DSP48s, etc.) and nets (input/output pins with optional routing elements). Modules, although found less frequently in conventional designs, are different in that they present a level of hierarchy in the design. A module contains a collection of instances and nets which are treated as a single entity. These modules are particularly useful to this work because they can be used to represent hard macro definitions which contain relative placement and routing information. Module instances are simply the instantiation of a module in a design.

One of the most challenging tasks in designing RapidSmith was to provide light-weight, fast and efficient files describing an FPGA device. This can be a difficult task where modern FPGAs can contain 10,000,000s of separate wires and over 100,000 primitives. In RapidSmith, this task has been accomplished by aggressive object reuse and custom serialization

[1]However, since that time, open source projects leveraging XDL called Torc [12] and OpenPR [13] have been developed.

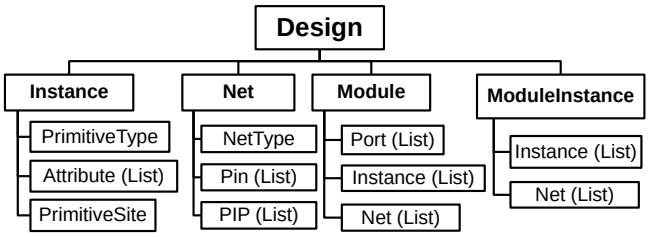

Fig. 2. Design Abstractions in RapidSmith Patterned After XDL

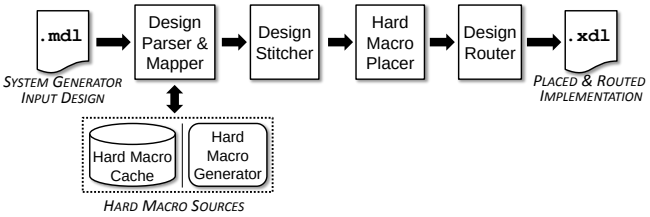

Fig. 3. Block Diagram of HMFlow

layered with a publicly-available serialization protocol [15]. Using this technique (see [16] for details), RapidSmith is able to achieve dramatic reduction of FPGA device representation by converting XDLRC report files into a format that is 10000-13000× smaller than the raw text. For example, a 23 GB XDLRC report file detailing a Virtex 6 LX760 part can be compacted into a file that is less than 2MB and loads into memory in less than 2.5 seconds. The size, and more importantly, the load times of these files were critical for the construction of a rapid prototyping tool flow.

Given our success of using RapidSmith to create custom FPGA CAD tools such as placers and routers, RapidSmith has been released as an open source tool for the benefit of the research community. RapidSmith downloads, documentation and examples are available at (removed for blind review).

IV. HMFlow: A Rapid Prototyping Flow

HMFlow is the vehicle by which we demonstrate the effectiveness of hard macros at accelerating FPGA compilation. The flow begins by importing designs created using Xilinx System Generator as seen in Figure 3. Design data are stored in Simulink Model Files (.MDL) that are parsed by the design parser and mapper tool. The mapper also identifies each block in a design and its corresponding hard macro. If the hard macro does not exist in the hard macro cache, the mapper invokes the hard macro generator to create one for the mapper 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. The design stitcher also inserts appropriate I/O buffers (IOBs) and clock generation circuitry. The design is then passed to the hard macro placer and router to be exported as a final placed and routed implementation in XDL.

The remainder of this section describes in detail the design entry techniques, algorithms and steps that have been con-

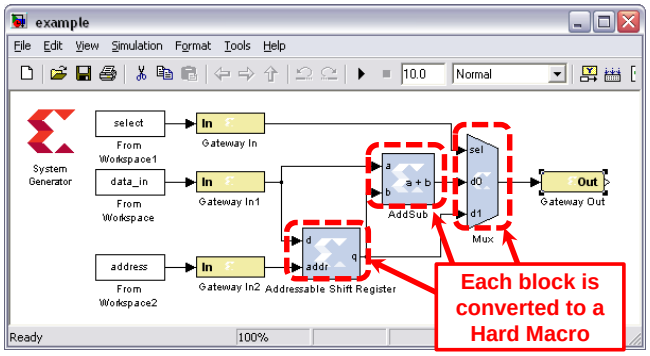

Fig. 4. Snapshot of an example System Generator Design

structed to realize and implement HMFlow.

A. Xilinx System Generator

Although HMFlow can be used with any design entry tool, System Generator was chosen for this work because it provided several benefits out-of-the-box. System Generator is well-suited to hard macros because it is a block-based design tool. HMFlow automatically converts each of the System Generator blocks into a hard macro as shown in Figure 4. Also, since HMFlow uses System Generator designs as input, existing System Generator simulation tools can be leveraged for HMFlow-bound designs.

Another advantage of using System Generator is its design compilation path. HMFlow provides a rapid prototyping path so designs can be quickly tested on an FPGA. However, once the design is functionally correct, a high quality implementation is needed for design deployment. Since HMFlow accepts designs that are System Generator compatible, a high QOR is easily obtained (albeit with much longer runtime) by simply using the conventional Xilinx System Generator tools.

B. Design Parsing, Hard Macro Mapping and Generation

1) Simulink Design Parser: The first step of HMFlow is to parse the System Generator design stored in the Simulink Model file (MDL). HMFlow has a custom built JavaCC-based parser to perform this task and populates a custom made Simulink-compatible data structure.

One drawback of using System Generator is that information about design blocks, such as bit widths of ports, are not stored in the MDL file. Therefore, the parser has to recalculate these widths on-the-fly during parsing and it must be performed on a block-by-block basis. Despite this technical challenge, HMFlow is able to support over 75% of the most commonly used System Generator blocks in HMFlow designs.

2) Hard Macro Mapping: The mapper receives the design from the parser and iterates over all of the blocks in the data structure to identify the System Generator blocks. For each block it finds, an MD5 hash is generated from its type and parameters to uniquely identify the block and corresponding hard macro. The mapper then uses the hash to check the cache to see if the hard macro has already been built. If the hard macro has already been built, it is loaded from the cache and given to the mapper. If the hard macro is not in the cache, the mapper invokes the hard macro generator.

3) Automated Hard Macro Generation: Xilinx does not provide any method for automated hard macro creation of arbitrary designs. The only method provided is a tedious, manual process using FPGA Editor, therefore, an automated hard macro generator was created using RapidSmith. This hard macro generator is an improved version of that used in preliminary work [1].

To create hard macros in HMFlow, arbitrary design blocks (NGC netlists from System Generator) are implemented with the conventional Xilinx tools. Designs are prepared by inserting special port-identifying hard macros which are attached to each of the inputs and outputs of a design. These special hard macros remain untouched as the design passes through the Xilinx tools and specify the ports to create in the final hard macro. After synthesis of the hard macro-bound design, resource utilization counts are extracted from the synthesis report to generate slice, DSP48 and BRAM placement constraints. These constraints ensure that hard macros are compact and also maximize the number of valid locations they can be placed on the FPGA.

Once the placed and routed design (NCD file) is generated with the Xilinx tools, it is converted to XDL for manipulation by the hard macro generator. The hard macro generator then performs various transformations[2] on the design to convert it to a Xilinx compatible hard macro. The hard macro is then stored in the hard macro cache using a similar compact format to that used in RapidSmith device files as described in Section III-B.

C. Design Stitcher

After the mapper has gathered all of the hard macros corresponding to a design, the design stitcher must perform the "stitching" of all the hard macros together into a single implementation. Design stitching requires the creation of nets in the XDL to designate connections between the hard macro ports. These connections are created based on connections present in the original System Generator design description.

Once all of the nets have been created, the design stitcher will instantiate the proper IOBs according to the pins designated in the System Generator design. Once these IOBs are created, clock generation circuitry is added to the design. In the Virtex 4 series, an additional step is required to avoid the NBTI effect [17] which requires the connection of all unused DCMs to an on-chip oscillator, the PMV primitive. After this step, the design is completely mapped to the FPGA and could be output to XDL. However, to avoid slow hard disk access times, the design is passed directly to the hard macro placer.

D. Hard Macro Placer

Since HMFlow uses hard macros instead of FPGA primitives as the objects that are placed onto the FPGA fabric, there is a significant reduction in problem size. Even though the hard macros produced from System Generator design blocks are

[2]More details on hard macros and their conversion can be found in RapidSmith documentation [16].

relatively fine-grained, they still result in a 10-20× reduction of objects requiring placement. However, the algorithm used to perform placement also affects how fast placement will occur. Throughout the development of HMFlow, three different placers were created, a recursive bi-partitioning placer, a random placer and finally a greedy heuristic placer.

1) Recursive bi-partitioning Placer: The most commonly used FPGA placement algorithm is simulated annealing. This is due to its ability to obtain high quality implementations (which are often characterized by minimized wire length) at the expense of longer runtimes. In most cases, this is quite desirable because of the need to obtain a high QOR. However with rapid compilation as a goal for HMFlow, different algorithms were tested for their ability to perform placement quickly.

Our first attempt at a fast placer followed the approach of Maidee et. al [18] which used a recursive bi-partitioning algorithm. This involves partitioning the hard macros of a design into two separate groups such that the number of connections between the two groups is minimized. The two groups were created and optimized using the heuristic developed by Kernighan and Lin [19]. By recursively dividing each subgroup in this manner, it would ultimately reduce total wire length of the nets in the design. Our attempts using this implementation did show significant acceleration of the placement process, however, the quality of most of the resulting implementations resulted in clock rates averaging approximately 67 MHz on Virtex 4 parts. This result may be due to our choice of partitioning parameters or the lack of a final simulated annealing step as used in [18].

2) Random Placer: Through various experiments, random placements of several hard macro designs were attempted. The results were surprising in that most of the designs were still route-able and produced similar implementation quality to that of the recursive bi-partitioning placer. The random placer was extremely fast, placing most designs in a fraction of a second. This was due to the fact that each hard macro was placed only once rather than being swapped several times as in the partitioning placer.

3) Heuristic Placer: The conclusion made from the surprising performance of the random placer is that rapid placement can be achieved by placing each hard macro once but doing so in a manner that made an attempt at optimizing the placement. This would improve on the quality of the random placement, but still retain the fast execution time of the random placer.

The greedy heuristic developed was quite simple since it depended mostly on the amount of connectivity between hard macros. The heuristic dictates that hard macros are placed one-by-one starting with those containing DSP48 or BRAM primitives as those are the most scarce primitives on the FPGA fabric. Following the DSP/BRAM hard macros, placement priority is given to the largest (greatest number of occupied tiles) hard macros first as they may be the next most difficult to place.

When a hard macro is placed, the 1st, 2nd, or 3rd most connected hard macro to the current hard macro is queried to see if it has already been placed. If so, the current hard macro is placed as close as possible to its highly connected companion. If none of the three hard macros are already placed, the hard macro is placed arbitrarily into one of nine bins designated on the FPGA fabric. This process proceeds until all hard macros are placed. This technique has proven successful on all benchmark designs used when resource utilization is 50% or less of the FPGA. Results comparing the three placement algorithms are deferred to Section V where the benchmark designs used are described in detail.

E. Full Design Router

The Xilinx router, `par`, has an option (`-p`) to only perform the routing of a design, however, if a design is partially routed, those routes are not guaranteed to remain intact. This is a problem if the output of HMFlow were to create a placed hard macro design in XDL and then converted to NCD for `par` to route because the hard macros are flattened in the XDL to NCD conversion process. This would negate any savings provided by hard macros which contained routed nets. To leverage pre-routed routes from hard macros, a custom, full design router was created using RapidSmith for HMFlow.

Traditionally, the most popular FPGA router algorithm has been PathFinder [20] because of its ability to negotiate routing conflicts in an effective way. However, PathFinder requires making several iterations over the nets of a design in order to complete successfully. It is this iterative nature that optimizes circuit speed at the expense of longer runtime. For this reason, a different algorithmic approach was taken for HMFlow.

A simple FPGA routing technique is that of a maze router and is the principle algorithm used in this work. A maze router is fast because it only makes a single pass through all the nets in a design. However, unlike PathFinder, FPGA routing resources are assigned on a first-come, first-served basis which can cause significant routing conflicts given certain scenarios.

To overcome these potential conflicts, the router in HMFlow uses a congestion avoidance technique. When a design is first loaded in the router, it is analyzed for potential conflicts (which are mostly architecture-specific) by looking for hot spot routing switch boxes or certain input pins that have a unique input path. In these cases, the routing resources are reserved for the net which will require them most to complete the route. This technique has been successful in all designs tested when FPGA slice utilization has been 50% or less. Mixed results are found when utilization exceeds half of the FPGA. Despite this limitation, the router used in HMFlow has proven to be 3-10× faster than the fastest configuration of the Xilinx router (`par -p`) execution[3] and allows for preservation of internal hard macro routes. This router is also capable of routing arbitrary designs and can be used for general purposes.

[3]Because of the limitation by which `par` can be timed for execution, the performance comparison represents the time it takes to load a design from disk, route the design and then save the routed design to disk for `par` and the router used in HMFlow. It is expected that if file access times were removed, the HMFlow router would be faster than this estimate.

TABLE I
BENCHMARK DESIGN CHARACTERISTICS

Benchmark Name	V4 Part Name	Slices Used	BRAMs Used	DSP48s Used	HMs* / HMIs**
pd_control	SX35	150	1	0	12/21
polyphaseFilter	SX35	680	8	4	30/79
aliasingDDC	SX35	806	1	3	25/78
dualDivider	SX35	1832	0	6	39/542
computeMetric	SX35	2551	56	40	64/332
fft1024	SX35	2553	8	12	48/313
filtersAndFFT	SX35	5203	25	31	92/588
frequencyEst	SX55	6988	31	72	249/757
dualFilter	SX55	11173	33	26	93/901
trellisDecoder	LX200	16973	61	53	196/1328
filterFFTCM	LX200	18883	81	12	149/920
multibandCorr	LX200	19732	52	23	90/1472
signalEst	LX200	23841	126	47	390/1448

*HMs = Unique hard macros in the design
**HMIs = Total hard macro instances in the design

TABLE II
HARD MACRO PLACER ALGORITHM COMPARISON

Benchmark Name	Runtime (seconds)			Clock Rate (MHz)		
	REC	RAND	FAST	REC	RAND	FAST
pd_control	0.266	0.047	0.016	92	65	129
polyphaseFilter	3.281	0.063	0.015	111	65	108
aliasingDDC	3.234	0.047	0.016	97	67	107
computeMetric	15.86	0.157	0.047	69	54	57
fft1024	11.703	0.094	0.047	67	47	74
frequencyEst	29.156	0.391	0.219	50	30	60
filterFFTCM	203.9	2.765	0.984	43	27	37
Averages	38.2	0.51	0.192	75.6	50.7	81.7

REC: Recursive bi-partitioning placer, RAND: Random placer, FAST: Fast heuristic placer

V. RESULTS

This section describes the efforts to ensure adequate benchmark designs that properly illustrate HMFlow's capability to scale to large designs and large commercial FPGAs. Using these benchmarks, comparison data of the three placement algorithms developed for HMFlow is put forth. Finally, comparison data of the benchmarks implemented with HMFlow and conventional Xilinx tools is presented and analyzed.

All data presented in this section was measured on an HP workstation running Windows XP SP3 32-bit, with an Intel Core 2 Duo 3.0 GHz processor and 4GB RAM. All FPGAs tested are from the Xilinx Virtex 4 series with a speed grade of `-10` and Xilinx ISE ver. 12.3 was used for all experiments. RapidSmith and HMFlow Java implementations used the Oracle JVM ver. 1.6.0_21.

A. Benchmark Designs

In order to adequately demonstrate the effectiveness of hard macros on FPGA compilation time and their potential for design size scalability, a wide variety of design types and sizes were compiled and used. These designs, outlined in Table I, contain a variety of algorithms, processing cores and data paths. Some of the smaller designs include circuits such as a PicoBlaze, FIR filters, polyphase filters, state machines and 1024-point FFTs/IFFTs. Most of the larger blocks were taken from a very large experimental telemetry receiver design [21] which required three large FPGAs for its implementation. The telemetry receiver design was an excellent source of benchmark circuits because of its large size and also because the design was originally implemented and optimized in System Generator.

As shown in Table I, three different Virtex 4 devices were targeted, the SX35, SX55 and LX200 which is the largest part in the series. The table also shows that the 13 benchmark designs range in size from 150 slices to over 23,000 slices. BRAM and DSP48 primitive counts are representative of the typical circuits used and are present to illustrate that HMFlow and hard macros can be used with heterogeneous types of primitives without issue. The final column of Table I shows the number of unique hard macros the design contains and also the total number of instances of hard macros in the design. The latter number illustrates the number of objects the placer must place whereas the former illustrates the number of hard macros that exist in the cache and must be created. In most cases, hard macros are reused on average between 2 and 10 times illustrating another reuse benefit of hard macros.

B. Hard Macro Placer Algorithms

As mentioned in Section IV-D, three placers were created during the development of HMFlow: a recursive bi-partitioning placer (REC), a random placer (RAND) and a fast heuristic placer (FAST). The subset of benchmark designs available during placer development were placed three times, once for each placer as shown in Table II. Clock rates were obtained after each placement was routed with the HMFlow router and measured with the Xilinx tool, `trce`.

As can be seen from the listed averages at the bottom of Table II, the FAST placer had the fastest execution of the three placers and produced, on average, the highest quality circuit. These results show that the custom heuristic designed for hard macro placement is capable of producing higher quality results than the RAND and REC placer yet still execute with speed similar to that of the RAND placer.

C. HMFlow Performance

After the complete realization of HMFlow, all 13 benchmark circuits were implemented with both the Xilinx tools and HMFlow. Since the benchmark circuits were created as Xilinx System Generator designs, the Xilinx tools runtime included the time elapsed during the following steps:

- System Generator NGC generation (outputs netlist)
- `NGDBuild` (outputs NGD)
- `map` (outputs mapped circuit in NCD format)
- `par` (outputs placed and routed circuit)

In the steps of the Xilinx flow where the execution could be optimized to reduce overall runtime, it was done so to ensure that the comparison is made with the fastest execution of the Xilinx tools to implement a placed and routed design. Although theoretically possible, bitstream creation (.NCD to .BIT) could not be legally accelerated by HMFlow due to

TABLE III
RUNTIME PERFORMANCE OF HMFLOW AND COMPARISON TO XILINX FLOW

Benchmark Name	Simulink Parser	Mapper / HM Cache	Design Stitcher	Hard Macro Placer	Design Router	XDL Export	HMFlow Runtime	Xilinx Runtime	HMFlow Speedup[1]	XDL to NCD
pd_control	0.093s	0.735s	0.187s	0.016s	0.219s	0.062s	1.3s	65.6s	50×	2.8s
polyphaseFilter	0.094s	0.75s	0.219s	0.015s	1.406s	0.11s	2.6s	60.3s	23.2×	4s
aliasingDDC	0.11s	0.765s	0.219s	0.016s	1.453s	0.125s	2.7s	62.2s	23.1×	7.4s
dualDivider	0.313s	0.89s	0.203s	0.047s	2.407s	0.218s	4.1s	96.6s	23.7×	6.3s
computeMetric	0.281s	0.891s	0.641s	0.047s	6.359s	0.609s	8.8s	160.8s	18.2×	17.1s
fft1024	0.235s	0.937s	0.297s	0.047s	4.953s	0.375s	6.8s	119.3s	17.4×	10.3s
filtersAndFFT	0.328s	0.984s	0.797s	0.188s	12.312s	0.75s	15.4s	254.1s	16.5×	20.3s
frequencyEst	0.437s	1.5s	0.578s	0.219s	18.11s	1.171s	22s	373.5s	17×	107.3s
dualFilter	0.469s	1.313s	1.203s	0.437s	34.672s	1.656s	39.8s	469s	11.8×	140.4s
trellisDecoder	0.656s	1.719s	1.422s	0.547s	54.015s	2.5s	60.9s	824.6s	13.5×	115.1s
filterFFTCM	0.516s	1.937s	1.641s	0.984s	69.938s	3.046s	78.1s	1021.3s	13.1×	541.2s
multibandCorr	0.828s	1.797s	1.844s	1.859s	73.297s	5.781s	85.4s	786.2s	9.2×	506.7s
signalEst	0.843s	2.328s	2.157s	1.531s	107.547s	15.375s	129.8s	1508.7s	11.6×	869.2s

[1] We define HMFlow speedup as the time it takes Xilinx to create a placed and routed implementation divided by the time it takes HMFlow to create a placed and routed implementation from the same System Generator design.

licensing issues, thus, it is omitted as the runtime of `bitgen` would be identical for either of the design flows.

The total runtime of HMFlow is measured as the sum of time elapsed during the following steps:

- Simulink Parser (reads/parses System Generator design)
- Mapper/Hard Macro Cache (maps/retrieves hard macros)
- Design Stitcher (combines hard macros together)
- Hard Macro Placer (places all hard macros)
- Design Router (routes all un-routed nets)
- XDL Export (outputs placed and routed XDL design file)

Table III contains the runtimes for the individual steps in HMFlow, the total runtimes of HMFlow and Xilinx tools and the speedup of HMFlow over the Xilinx tools. A preliminary observation is that the speedup obtained by HMFlow is much greater (23-50×) for the smaller designs than the larger designs (9.2-13.5×). There are a few possible causes for this phenomenon. First, the Xilinx tools write out intermediate design files between each step of the design process and must also load device database files for each step. This is in contrast to HMFlow which loads design information once (during Simulink parsing and hard macro cache accesses) and loads the RapidSmith device files once for the entire execution of HMFlow. Therefore, the Xilinx tools spend a disproportionate amount of time reading/writing files for the smaller designs. A second possible cause for the faster speedup of smaller designs could be attributed to the algorithms used in the Xilinx tools. As Xilinx must accommodate very large designs on very large parts, their algorithms might be tuned for larger designs and thus, perform better. However, our suspicions lie with the former explanation rather than the latter.

If the runtimes of all benchmarks are averaged, Figure 5a shows a pie chart illustrating the runtime distribution of HMFlow. As can be seen from Figure 5a, the majority of the time is spent in the router. This illustrates the speed at which all the other steps in HMFlow operate. A completely placed implementation can be obtained from HMFlow in a matter of seconds for even the largest benchmarks. Although the RapidSmith router runs 3-10× faster than the Xilinx router, it is still the most time consuming process in HMFlow. This

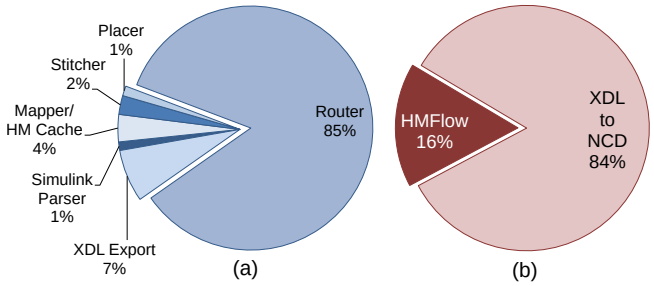

Fig. 5. (a) Average runtime distribution of HMFlow (b) HMFlow Runtime as a percentage of total time to run HMFlow and create an NCD file

is likely because of the granularity of hard macros used in the benchmark designs. The hard macros do not contain a significant percentage of all nets in the final design, most of the nets exist between the external ports of the hard macros. For this reason, the router must route almost all of the nets in the design in the final step of HMFlow. Future work on HMFlow will concentrate on creating larger, more routing-dense hard macros to help alleviate this problem.

One technical issue with HMFlow and the Xilinx tools is the time it takes to create an NCD file from an XDL file. The file formats are equivalent, however, Xilinx `bitgen` only accepts as input NCD files and thus, in order to create a bitstream from HMFlow, an NCD file must be created. The far right column of Table III lists the runtime to generate an NCD file for the resulting HMFlow benchmark implementation. As designs get larger, the conversion time escalates. Figure 5b shows the average runtime distribution if the runtime of HMFlow and XDL conversion are combined. It is quite puzzling that HMFlow can parse, assemble, stitch, place and route a design in less than 1/5 the time it takes the Xilinx tools to convert that implementation into a different format. However, this illustrates the great efficiency which HMFlow has achieved and the speed of implementation that can be had for FPGA compilation.

In order to achieve the dramatic reduction in runtime, HMFlow trades faster runtime for a lower maximum clock

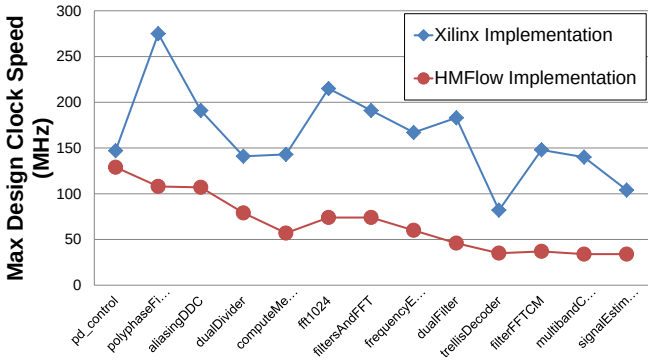

Fig. 6. A comparison plot of the benchmark circuits maximum clock rates when implemented with HMFlow and the Xilinx tools (the reader is reminded that the Xilinx tools are likely to have produced a higher quality circuit, however, these figures represent the fastest Xilinx tools configuration).

rate as shown in Figure 6. On average, HMFlow produces implementations that are 2-4× slower than that of Xilinx. For rapid prototyping purposes, this tradeoff is acceptable because even though the design will run 2-4× slower, overall, it will be executing 10,000s of times faster than simulation.

VI. Conclusion

The goal of this paper is to demonstrate the effectiveness of using hard macros to reduce FPGA compilation runtime for rapid prototyping. When a designer can sacrifice 2-4× in clock rate and limit FPGA utilization to 50%, HMFlow is capable of compiling designs 10-50× faster than the fastest Xilinx flow. This ultimately translates into improved productivity for debug and verification engineers. HMFlow is able to accomplish this by leveraging non-traditional algorithms and a new XDL framework for Xilinx FPGAs called RapidSmith.

Although HMFlow has been shown to reduce compilation runtime by an order of magnitude or more, the long XDL to NCD conversion time will ultimately limit the speedup obtained. Even though efforts are being made to address this issue, future work on HMFlow will focus on rapid compilation for high QOR implementations. By focusing on high QOR, XDL to NCD conversion time will become a smaller fraction of the runtime in high QOR compilation. Hard macros will continue to play an integral part of the new approach as they have the capacity to capture timing closure information for larger design blocks which ultimately can be reused from run to run. HMFlow will also expand to include support for newer Xilinx devices such as Virtex 5. We also plan to enable HMFlow to accept as input designs created in other designs tools besides System Generator.

References

[1] C. Lavin, M. Padilla, S. Ghosh, B. Nelson, B. Hutchings, and M. Wirthlin, "Using Hard Macros to Reduce FPGA Compilation Time," in *Proceedings of the 20th International Workshop on Field-Programmable Logic and Applications (FPL'10)*, August 2010.

[2] P. K. Chan and M. D. F. Schlag, "New Parallelization And Convergence Results For NC: A Negotiation-Based FPGA Router," in *FPGA '00: Proceedings of the 2000 ACM/SIGDA eighth international symposium on Field programmable gate arrays*. New York, NY, USA: ACM, 2000, pp. 165–174.

[3] Y. Xu and M. Khalid, "QPF: Efficient Quadratic Placement For FPGAs," in *Proceedings of the IEEE International Conference on Field-Programmable Logic and Applications. IEEE, Los Alamitos, CA*, 2005.

[4] V. Betz and J. Rose, "VPR: A New Packing, Placement And Routing Tool For FPGA Research," in *Proceedings of the 7th International Workshop on Field-Programmable Logic and Applications*. Springer-Verlag London, UK, 1997, pp. 213–222.

[5] C. Mulpuri and S. Hauck, "Runtime And Quality Tradeoffs In FPGA Placement And Routing," in *Proceedings of the 2001 ACM/SIGDA ninth international symposium on Field programmable gate arrays*. ACM New York, NY, USA, 2001, pp. 29–36.

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

[7] Y. Sankar and J. Rose, "Trading Quality For Compile Time: Ultra-Fast Placement For FPGAs," in *Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*. ACM New York, NY, USA, 1999, pp. 157–166.

[8] J. S. Swartz, V. Betz, and J. Rose, "A Fast Routability-Driven Router For FPGAs," in *FPGA '98: Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays*. New York, NY, USA: ACM, 1998, pp. 140–149.

[9] 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.

[10] 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: IEEE Computer Society, 2008, pp. 211–216.

[11] 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*, ser. CODES/ISSS '10. New York, NY, USA: ACM, 2010, pp. 13–22.

[12] N. Steiner, A. Wood, H. Shojaei, J. Couch, P. Athanas, and M. French, "Torc: Towards an Open-Source Tool Flow," in *Proceedings of the 19th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '11. New York, NY, USA: ACM, 2011.

[13] A. A. Sohanghpurwala, "OpenPR: An Open-Source Partial Reconfiguration Tool-Kit for Xilinx FPGAs," Master's thesis, Virginia Tech, December 2010.

[14] 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.

[15] S. Ferguson and E. Ong, "Hessian 2.0 Serialization Protocol," http://hessian.caucho.com/doc/hessian-serialization.html, August 2007.

[16] C. Lavin, M. Padilla, P. Lundrigan, B. Nelson, B. Hutchings, and M. Wirthlin, "RapidSmith: A Library for Low-level Manipulation of Partially Placed-and-Routed FPGA Designs," Brigham Young University, http://rapidsmith.sourceforge.net, Tech. Rep., 2010-2011.

[17] A. Lesea and A. Percey, "Negative-Bias Temperature Instability (NBTI) Effects in 90 nm PMOS," Xilinx Inc., http://www.xilinx.com/support/documentation/white_papers/wp224.pdf, White Paper 224, November 2005.

[18] P. Maidee, C. Ababei, and K. Bazargan, "Fast Timing-driven Partitioning-based Placement for Island Style FPGAs," *Design Automation Conference*, vol. 0, p. 598, 2003.

[19] B. W. Kernighan and S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs," *The Bell System Technical Journal*, vol. 49, no. 1, pp. 291–307, 1970.

[20] L. McMurchie and C. Ebeling, "PathFinder: a Negotiation-based Performance-driven Router for FPGAs," in *Proceedings of the 1995 ACM Third International Symposium on Field-programmable Gate Arrays*, ser. FPGA '95. New York, NY, USA: ACM, 1995, pp. 111–117.

[21] 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*, July 2008, pp. 250–256.