BPR: Fast FPGA Placement and Routing Using Macroblocks

James Coole and Greg Stitt
Department of Electrical and Computer Engineering
University of Florida
Gainesville, FL 32611
{jcoole, gstitt}@ufl.edu

ABSTRACT

Numerous studies have shown the advantages of hardware and software co-design using FPGAs. However, increasingly lengthy place-and-route times represent a barrier to the broader adoption of this technology by significantly reducing designer productivity and turns-per-day, especially compared to more traditional design environments offered by competitive technologies such as GPUs. In this paper, we address this challenge by introducing a new approach to FPGA application design that significantly reduces compile times by exploiting the functional reuse common throughout modern FPGA applications, e.g. as shared code libraries and unchanged modules between compiles. To evaluate this approach, we introduce Block Place and Route (BPR), an FPGA CAD approach that modifies traditional placement and routing to operate at a higher-level of abstraction by pre-computing the internal placement and routing of reused cores. By extending traditional place-and-route algorithms such as simulated-annealing placement and negotiated-congestion routing to abstract away the detailed implementation of reused cores, we show that BPR is capable of orders-of-magnitude speedup in place-and-route over commercial tools with acceptably low overhead for a variety of applications.

Categories and Subject Descriptors

J.6 [**Computer-aided Engineering**]: Computer-aided Design

General Terms

performance, design

Keywords

macroblock, placement and routing, core, reuse, coarse-grain, FPGA, speedup

CODES+ISSS'12, October 7-12, 2012, Tampere, Finland.

1. INTRODUCTION

Existing work has shown the performance and power advantages of hardware/software co-design using FPGAs [1, 2]. However, the complexity of current FPGA design methodologies has been a barrier to mainstream adoption of this promising technology [3]. Chiefly, typical placement and routing (PAR) times of tens of minutes to hours and even days [4] present significant challenges in the development of FPGA systems by lengthening the design-debug cycle and decreasing the turns-per-day of designers, resulting in decreased design quality and increased time to market. FPGAs also increasingly compete for the attention of designers against technologies without such problems, such as DSPs and GPUs, which even if not ideal in terms of performance, have compile times of seconds to minutes. In fact, increasingly popular languages such as OpenCL rely on these fast compile times for runtime compilation, which prevents fully compliant OpenCL usage on FPGAs. Furthermore, ever larger designs making use of rapidly increasing device resources means that PAR times will only continue to grow. Because this trend does not hold for competing technologies such as CPUs and GPUs, without a new approach, today's FPGA productivity gap will only be exacerbated by tomorrow's devices.

In this paper, we present an approach for significantly reducing PAR times based on functional reuse. Many modern FPGA designs often incorporate custom datapaths for efficient computation as part of a larger system on chip (SoC) that includes large cores such as soft processors and interfaces to external board components. Because their design and verification is inherently time consuming, SoC cores are good opportunities for reuse and are often selected by the designer from shared or vendor libraries such as Altera's SOPC builder [5] or OpenCores [6], or are imported from previous designs. Custom datapaths, too, often make use of widely used cores such as floating-point arithmetic operators. Even in cases where truly unique cores must be developed for a single instance in an application, the core is ultimately reused in time, present in numerous compiles over the course of the system's development and testing. Our approach to placement and routing allows this existing reuse to enable large PAR speedups for most designs.

Towards this end, we've developed a new FPGA CAD approach, Block Place and Route (BPR), that modifies traditional FPGA placement and routing to operate at a higher level of abstraction. Unlike traditional FPGA CAD flows, shown in Figure 1(a), which decompose a circuit into numerous fine-grained lookup-tables (LUTs) during every compi-

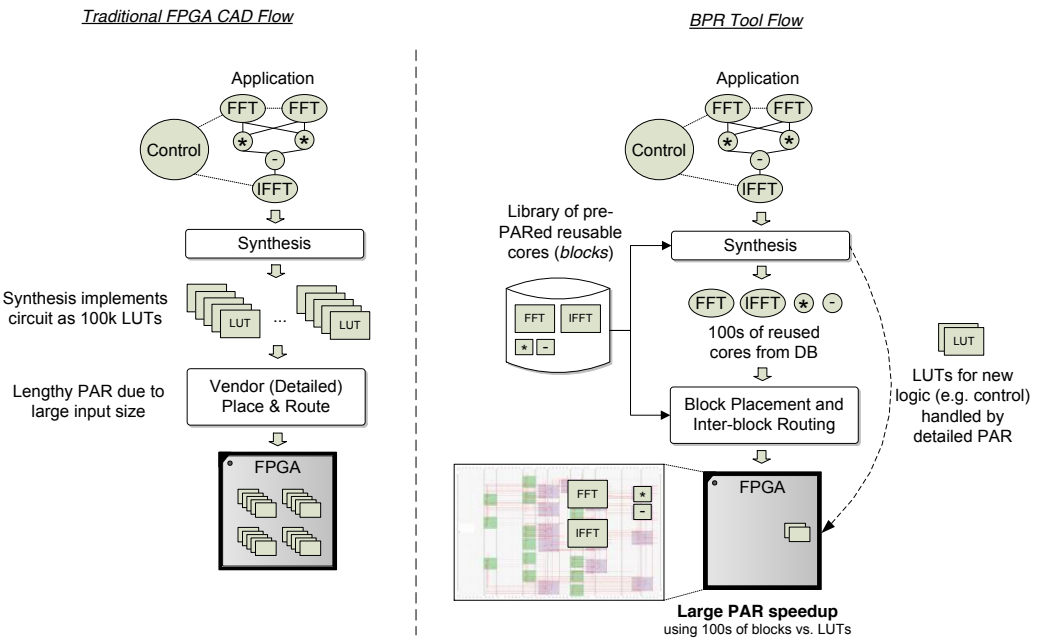

Figure 1: (a) Traditional CAD flows decompose circuits into numerous LUTs, resulting in lengthy place and route. (b) BPR enables fast placement and routing of reused cores with a database of pre-computed core implementations for different FPGA locations. For each reused core, BPR only has to place the black-box footprint (i.e. *macroblock*) of the core, and routes only the connections between macroblocks, providing 100x speedup compared to commercial tools.

lation, BPR pre-computes the internal placement and routing of reused cores for numerous positions in the FPGA. As shown in Figure 1(b), BPR enables reused cores to be compiled a single time into a core database and then reused in different locations on the FPGA in subsequent compiles, with only new logic requiring a full-detail PAR. We extend traditional PAR algorithms such as simulated-annealing placement [7] and negotiated-congestion routing [8] to place only the footprints of these pre-computed cores, called *macroblocks*, and route only the communication between cores. This abstraction achieves large speedups in PAR up to 190x over commercial tools, enabling average 1.1s compile times in line with competing technologies such as GPUs. By extending the best current approaches to traditional placement and routing, BPR achieves these speedups with acceptable overhead in terms of area (up to 49% utilization) and performance (average 34% decrease in clock frequency) relative to full-detail PAR performed by commercial tools for many applications. For situations where such overhead is prohibitive, BPR could speedup debugging iterations and be replaced with a full-detail PAR upon completion. Furthermore, BPR achieves these fast compilation times while using under 25MB for a large Altera EP3C120 Cyclone III, compared to up to 500MB for commercial tools. This small memory footprint potentially enables just-in-time (JIT) compilation of FPGA circuits for languages such as OpenCL.

2. PREVIOUS WORK

One previous method of accelerating PAR uses specialized devices that substitute fine-grain FPGA resources with coarse-grain functionality specific to various domains [9, 10]. This specialization can result in substantially faster synthesis and PAR through large reductions in problem size. Specialized devices can additionally achieve higher performance with reduced area through optimized hardware, but sacrifice the flexibility provided by the fine-grain reconfigurability of FPGAs. These specialized devices also require costly design and manufacture without the economies of scale of more general architectures.

Intermediate fabrics [11] and similar approaches [12, 13] achieve the PAR acceleration of coarse-grain physical architectures on commercial-off-the-shelf (COTS) FPGAs through virtualization, implementing the specialized architecture as an overlay network. Virtualization can retain some of the flexibility of the underlying FPGA by switching between any number of virtual fabric instances, each designed for different application domains. One limitation of intermediate fabrics is the area and clock (alternately, latency) overhead caused by implementing the architecture's routing resources on top of the underlying FPGA. Also, although fabric selection can be deferred to increase flexibility, the fabric instances used must still be designed offline and *a priori*, since the fabric itself must be compiled for the target device, typically requiring hours. This limitation leads inevitably to some additional overhead due to functional mismatches between a design and the available fabrics, and makes intermediate fabrics impractical for SoC-style design, where it isn't possible to incorporate every possible interface or large core *a priori*. BPR similarly uses abstraction to achieve PAR speedup, but has less overhead and does not have these flexibility limitations. BPR can also be complementary to intermediate fabrics, potentially improving the flexibility of intermediate fabrics by allowing fabric instances to be gen-

erated rapidly from precompiled functional units and fabric resources.

Other approaches have looked at modifying PAR algorithms to reduce runtimes at the expense of some quality. Lysecky introduced JIT FPGA compilation [14] and dynamic FPGA routing [15] to speedup place and route. However, that work required a specialized FPGA fabric and didn't directly support COTS FPGAs. Mulpuri and Hauck [16] explored routing quality and execution time tradeoffs, enabling up to 3x PAR speedups with 27% clock overhead. Although this speedup is significant for many uses, potential applications such as JIT compilation require larger speedups. Wires on Demand [17] introduced a fast PAR approach for partial reconfiguration (PR) flows, enabling rapid synthesis of interconnect between pre-PARed modules in a PR system. BPR is conceptually similar, but doesn't focus on statically planned PR regions, and might be used in conjunction with Wires on Demand to quickly synthesize additional modules whose performance or area is not of chief concern.

Previous work has also looked at reducing placement and routing times, or both, through the use of precompiled logic, called *macroblocks* or *hard macros*, by analogy with ASIC floorplanning approaches. Tessier [18] proposed using internal placement data for precompiled cores to speedup placement, using a multiple-step custom floorplanning and placement algorithm, achieving a total 2.6x speedup over commercial PAR times. Our approach also attempts to reuse internal placement data, except that we use modified versions of traditional simulated annealing FPGA placement algorithms. We also attempt to reuse pre-computed internal routing, resulting in substantially larger PAR speedups.

Most recently, Lavin et al. introduced HMFlow [19], a tool that makes use of both placement and routing information for precompiled cores, achieving 10x speedups over commercial PAR, with a 2-4x decrease in clock rate for designs up to 50% of device area. The current version of HMFlow focuses on smaller cores than BPR, using case studies with an average of 16 slices per core. HMFlow also uses a custom greedy-based placement heuristic and a single-pass congestion-avoiding router. BPR focuses on larger cores, including those encountered in SoC design and floating-point computations. Using larger cores can permit greater speedups, but introduces complexities which we discuss in this paper, including leaking routes and core packing/performance tradeoffs. BPR also uses modified versions of traditional and (at least theoretically) higher quality placement and routing algorithms, contributing to comparatively lower overhead in many cases.

Finally, many commercial FPGA toolchains include some support for manual incremental compilation and manual floorplanning, with a goal of avoiding re-synthesizing or recomputing the placement and routing of unchanged modules in a large design [20]. BPR accelerates compilation similarly, but instead pre-computes the PAR for each module in the design at many (typically all) possible locations on the device. This data gives BPR the ability to adjust to even large changes in communication between core instances in a design by essentially re-floorplanning the design, automatically, during BPR's placement phase. Note that BPR's more dynamic approach is more similar to the flexible placement of hard macros in ASIC design flows. One difficulty in adapting this technique for use with FPGAs is accommodating variations in the logic and routing resources across modern FPGAs, which is discussed in more detail in Section 3.2.

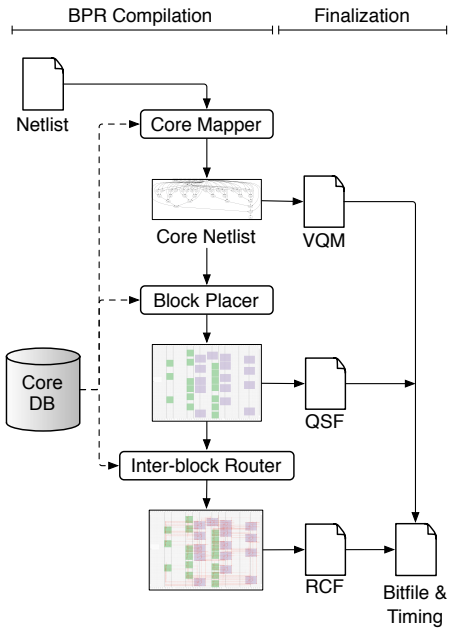

Figure 2: Overview of BPR tool flow. High-level BPR netlists are matched against relocatable core data, including intra-block PAR and port specifications. BPR then places each core as blocks and stitches blocks together through inter-block routing. The output files include an Altera Quartus-specific netlist file (VQM), placement (QSF) and routing (RCF) data for the full design. This design is validated and final bitfiles and timing are produced by Altera's Quartus.

3. BPR: MACROBLOCK-BASED PAR

This section describes the Block Placement and Routing (BPR) tool. Section 3.1 presents an overview of the tool flow. Section 3.2 explains how cores are integrated with BPR and converted into relocatable macroblocks. Section 3.3 discusses the BPR placement algorithm. Section 3.4 explains BPR routing.

The current version of BPR is compatible with Altera Cyclone III devices [21], so much of the discussion involves terminology specific to that vendor's devices. However, this practical limitation of our tool is due to the limited public information on commercial FPGA architectures. Note that BPR could be made to work for any island-style FPGA architecture with directional routing, given information about the device's layout and routing connectivity, and with the addition of any PAR data formatters appropriate for the platform. To our knowledge, BPR is the only non-proprietary router that can target Altera devices.

3.1 Overview

From the highest level, BPR performs placement for a complete design using black-box footprints of pre-implemented, coarse-grain cores, referred to as *macroblocks*. Such *block placement* is possible because every macroblock is relocat-

able to potentially any location on the FPGA, due to a BPR pre-processing step that computes PAR for the internals of the corresponding core at each block location, which we refer to as *intra-block PAR*. After placing each macroblock, BPR performs *inter-block routing* by stitching together the internal placement and routing of each placed macroblock. Speedup in placement and routing is realized by using coarse-grain cores, which reduces the problem size of PAR by ignoring the details of intra-block PAR, and focusing instead on how the system of macroblocks fit together on the device. The BPR tool flow that implements this approach is illustrated in Figure 2 and summarized in the following subsections.

3.1.1 Core Database (DB) and Mapping

BPR must initially be provided with a library of precompiled cores to stitch together to compose larger designs. The contents and construction of this database will be discussed more in Section 3.2, but it contains three main pieces of data needed by BPR: the name and port specification of each core, the rectangular size and origin of all legal placements of each core on the device, and the specific device resources (logic and routing) used by the core for each legal placement.

BPR accepts a simple high-level netlist format that supports core instances and nets capable of connecting arbitrary subranges of instances' ports. BPR performs *core mapping* on these netlists, similar to functional mapping in traditional FPGA tools, by matching instance types to the core types available in the core database, producing a legal *core netlist*. The mapper also takes care of miscellaneous issues such as clock net promotion (BPR currently uses Quartus to route clock signals). Although there are interesting opportunities for optimization of these high-level netlists, such optimizations are outside the scope of this paper. Furthermore, as shown in Figure 1(b), BPR can support a mixture of reused cores (high-level BPR netlists) and non-reused HDL handled in a post-processing step of traditional PAR (e.g. allowing the specification of unique control logic). However, in this paper we solely evaluate reused cores. For Altera target devices, BPR exports a core netlist to Quartus as a two-level hierarchical netlist for the complete design as VQM files specifying the top-level entity and core implementations.

3.1.2 Block Placement and Inter-block Routing

Once a core netlist has been created by the mapper, BPR's block placer attempts to choose between all legal block placements for each core (specified in the core database) in a way that fits all core instances on the device while minimizing the distance and expected routing congestion between connected instances. Note that a legal block placement for one core type will typically overlap many legal placements for other core types (and even other placements of the same type). Our approach to block placement uses a modified version of the VPR [7] simulated-annealing placement algorithm that we have extended to handle these issues. We provide a more detailed discussion of the block placer in Section 3.3. For Altera targets, the placement of each resource is exported as full placement constraints in a QSF file.

Finally, once the blocks for all core instances have been placed on the device, the inter-block router is only responsible for handling the nets connecting blocks because all intra-block routing can be loaded from the core database. First, the router must load from the core database and mark as unavailable all the routing resources utilized internally by each block. The router then loads the resource endpoints of each block's ports. Given these hard constraints, the nets between blocks are routed while minimizing route length. Our inter-block routing approach, discussed in more detail in Section 3.4, is a negotiated-congestion router based on PathFinder [8]. For Altera targets, BPR exports these inter-block routes, along with the internal routes of each block, as full wire-level routing constraints in an RCF file.

3.1.3 Bitfile and Timing

BPR provides the various output files shown in Figure 2 to Quartus for validation and bitfile creation. In the current version of BPR, Quartus is responsible for two other tasks due to limited available information for the Cyclone III device architecture. First, although BPR has information about the connectivity of most routing resources on Cyclone III devices, it does not have information about the device's clock distribution networks. Thus, we currently rely on Quartus to handle clock distribution. Second, for block PAR, the maximum clock frequency f_{max} is determined by two factors: the minimum clock rate of any of the core placements used in the final design and the maximum delay of the inter-block routes. The core database contains block timing data; however, BPR lacks detailed delay information for the Cyclone III's routing resources, currently requiring Quartus to compute the inter-block delays.

3.2 Core DB: Making Blocks Relocatable

As previously discussed, cores supported by BPR must be supplied in a pre-existing core database. BPR is capable of adding arbitrary cores to this database, specified as standard VHDL or Verilog modules. In this section we describe how the core database is populated and the process used to create the relocatable macroblocks used by the placer and router. This process is illustrated in Figure 3. Rather than performing these steps manually, we have created a collection of Python tools that integrate with external programs such as Altera Quartus for intra-block PAR and Synopsys Synplify for core synthesis.

3.2.1 Specification

Cores are added to the database by providing two sets of files to BPR: the standard VHDL or Verilog specification of the core module and an XML-based core specification file, *corespec*, that describes how the core should be handled by the tools. The corespec contains the core's name for use in netlists, a human-readable description for tools and debugging, and identifies the implementing HDL files and top-level entity name. The corespec also identifies and describes the ports that should be exposed in BPR, and allows specifying generics for parameterized modules. Cores may use multiple HDL files in their implementation, but must only contain a single top-level entity.

3.2.2 Synthesis and Resource Determination

The BPR database tools first wrap the core entity by a top-level interface enforcing the corespec. The wrapped entity is then synthesized using standard HDL synthesis tools, currently either Quartus or Synplify. This synthesis pass is required for future steps, but also provides standard optimizations for logic inside the module. Note that because

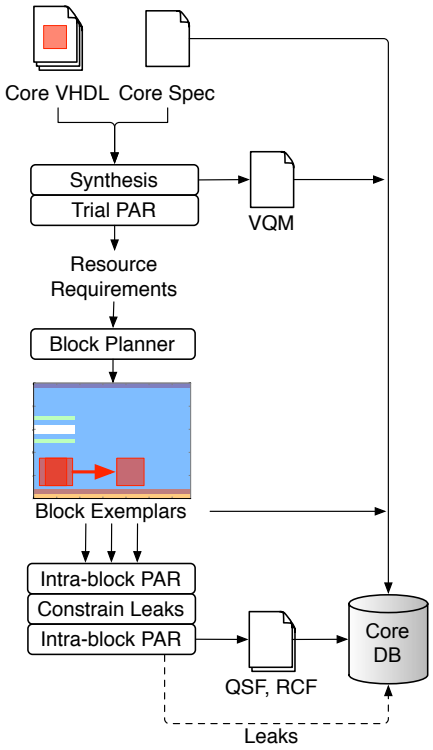

Figure 3: Overview of the BPR core database generation tool flow. New cores are added to the database using their HDL source files and a *corespec* file describing the core's generics and port structure. The tools plan possible placements of cores using black-box footprints (i.e. *blocks*) based on the core's resource requirements and then perform the intra-block PAR using vendor tools, taking care to remove *leaking routes* outside the block area.

the top-level entity is synthesized in this stage in isolation, optimization between cores is prevented, potentially reducing the quality or fit of the overall circuit. This limitation could be addressed through the use of high-level optimizations, but such approaches are likely to be domain-specific, and are outside the scope of this paper. Finally, the results of synthesis are cleaned up to remove entities inferred by Quartus, such as IO and clock buffers, and stored in the database for future export.

The core is then placed and routed on the device using Quartus, with top-level ports assigned to device pins. This *trial PAR* pass is used to find the device primitive types and counts required by the module, enumerating the device primitives used through back-annotating Quartus' placement to QSF. This method works even for cores incorporating protected IP (e.g. some Altera Megafunctions), where the primitives used would otherwise by obscured behind black-box instances in the VQM after synthesis.

3.2.3 Block Planning

Once Quartus or Synplify has optimized the core logic and determined the core's resource requirements, the BPR database tools can begin planning potential placements of blocks on the device. BPR is currently limited to handling

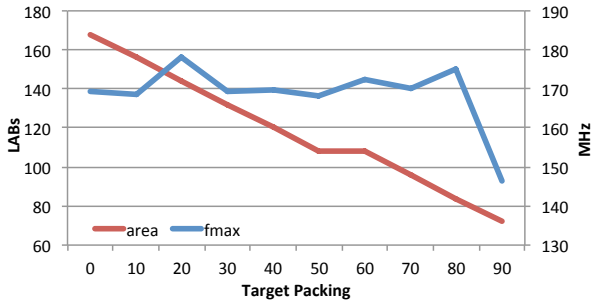

Figure 4: Packing tradeoffs for a floating-point adder core. Depending on the core's internal routing pressure, f_{max} decreases for smaller block sizes (i.e. higher packing ratios).

cores shaped as rectangular blocks. On Altera devices, we measure block geometry in terms of the coordinates of island structures such as the Logic Array Blocks (LABs) and special resources like M9Ks memory units [21].

The first step of block planning is determining block size. For cores composed of mostly logic (i.e. flip-flops and LUTs), block size is determined by the number of resources required for the core and the packing ratio of the LABs (i.e. the average ratio of flip-flops or LUTs occupied in each LAB). Low packing ratios waste device resources, as the placer can't resolve placement of the fine-grain structures within overlapping blocks. For large cores, very high packing ratios will sometimes fail intra-block PAR using Quartus, typically due to routing pressure within the core. However, even before intra-block PAR fails outright, high packing can reduce the f_{max} Quartus achieves for the block. In practice, choosing an appropriate packing ratio, and thus block size, is currently performed manually, with target ratios of 80-95% providing good area utilization without overly impacting performance. An example of these trends is shown in Figure 4. Note that the actual area used by the core (red line), set through the target packing ratio, changes in steps, depending on the closest rectangular area and other constraints on shape including the device's spacing for any required special resources. As the figure illustrates, for many cores, packing can be increased up to a point without significantly affecting f_{max}, but dropping rapidly after (e.g. 80% packing).

The second step of block planning involves identifying rectangular regions on the device that contain the necessary number and type of resources. To determine legal placements, the block planner uses a sliding-window approach, moving a block-sized footprint across the device one unit at a time while checking whether the resources covered are sufficient, as illustrated in Figure 3.

Once all the legal placements have been identified, the database tools group the placements into sets based on portability. The internal placement and routing of one block is portable to another location only if the logic and routing resources used inside the block exist at the same relative locations (and have the same connectivity) at the other location. The block planner determines portable sets by assigning signatures to each placement that incorporate information about the logic and routing resources within each placement footprint. From each of these sets, one block is chosen as a representative, called a *block exemplar*, which reduces

the work in subsequent stages by serving as a stand-in replacement for all the blocks in the set. The block planner ultimately stores a grid of all legal placements in the database, with each placement identifying its exemplar block, for the later use of the BPR placer.

3.2.4 Intra-block PAR and Leaking Routes

Finally, BPR determines the intra-block PAR of each exemplar block by another pass through Quartus. BPR can stitch together the intra-block PAR of blocks without conflict only if all the placed primitives and routes are contained within the block's footprint. For Altera targets, intra-block placements are confined to the rectangular placement footprint by specifying a LogicLock floorplanning constraint [22] on the top-level module. Unfortunately, current versions of Quartus don't guarantee that intra-block routing will be confined to the LogicLock region, so the output of this second pass often includes internal routes making use of resources outside the block footprint, which we call *leaked routes*.

Though we might wish LogicLock could take care of these issues for us, the inability of LogicLock to make guarantees about routing reflects the limitations of modern FPGA routing architectures. The general-purpose routing of modern FPGA architectures relies heavily on *long wires*, wires spanning multiple islands before terminating in a switch box [23]. For example, the primary general-purpose routing resources on the Cyclone III architecture are the R4 and C4 wires, which span four LABs in row and column channels, respectively. Length-16 and even -24 (C16 and R24) are available on the large devices in the Cyclone III family. These long wires complicate the confinement of intra-block routing. For example, for a 2x2 block footprint, no length-4 wires starting within the block can terminate within the block. Though local wires can usually handle the routes for such small blocks without leaks, in general, the more logic contained within a core and the smaller its block footprint, the more routes will leak. The effects and trends of these leaks are illustrated in Figure 5 for an example core, showing that, due to internal routing pressure and the use of long tracks, an increasing number of a core's internal routes leak from the core's placement footprint (# of LABs) as the size of the footprint is reduced. In the future, as FPGA device sizes continue to increase, we can expect an even greater reliance on long wires to keep interconnect delays low [23], meaning more leaked routes complicating intra-block PAR.

In most cases, there are two factors leading to leaked routes. First, for some nets, a leaking route may provide lower delays than any non-leaking route, helping achieve a high f_{max} for the block. Second, at high routing pressures it may not even be possible to route a net without leaking. Thus, BPR deals with leaked routes in two stages. First, any leaking routes are detected after the first pass through Quartus. The worst leaks are then constrained using additional routing constraints (RCF interval constraints) to the block footprint, and Quartus is re-run with all other routes locked. This process is effective to remove a large number of leaks, at some cost to f_{max} increasing with the number of routes re-routed. However, for large blocks, it is rare that all leaks can be fixed in this way. In this case, we currently add padding to the block's footprint to avoid overuse of the leaked routes' resources.

It should be noted that capturing the intra-block PAR for

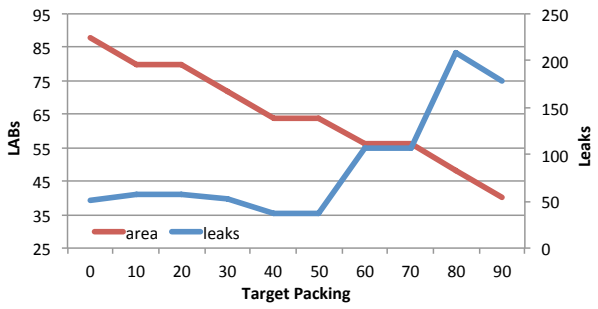

Figure 5: Number of leaked routes for different packing ratios for a floating-point multiplier. The number of routes extending outside the LAB block area (*leaking*) increases with larger packing ratios due to internal routing pressure.

all the legal placements of a core is typically the most time-consuming step involved in populating BPR's core database. For a large device, most cores will yield hundreds of exemplars after block planning, requiring hundreds of runs through Quartus, which can take considerable time. The time required to add a new core can be substantially reduced in two ways. First, although a core may have many exemplars, each exemplar is not equally valuable in terms of placement flexibility, with some exemplars corresponding to only a single placement option, and others covering tens or hundreds. Thus, we have added an option to our tools to *sift* exemplars, throwing out low-value exemplars until flexibility is impacted to some degree, specified as a percentage. Second, the intra-block PAR of each block placement is independent, and thus is computed in parallel by our tools (using up to one instance of Quartus per machine thread). For some use-cases, for example traditional offline FPGA design, this process could be parallelized over multiple machines making core addition extremely fast.

3.3 Block Placement

Given a legal core netlist, BPR's first task is determining where each core instance should be placed on the device. BPR's block placer is responsible for fitting all cores on the device by choosing between legal placement blocks, while considering routability by minimizing the distance between communicating instances and minimizing the expected congestion experienced by those nets.

Because of the similarities with traditional FPGA placement, we based our block placer on the well-known simulated-annealing placement strategy of VPR [7]. However, block placement differs significantly from traditional FPGA placement in two important ways: 1) blocks placed by BPR are not atomic in size and type, conflicting through overlap with multiple other placements, and 2) conflicting with placements of other core types. In this section we discuss how we modified VPR's placement algorithm to handle these complexities and support block placement.

Like VPR, our block placer begins with an initial, poor (randomly generated) placement and evolves an improved solution through many iteratively applied random adjustments, with a tendency to prefer improvements based on an annealing schedule. The solution is perturbed in each iteration by randomly selecting a block instance, placed or un-

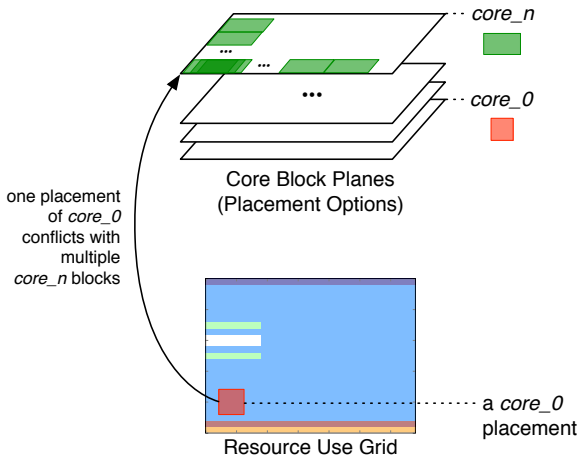

Figure 6: Placer conflicts and bookkeeping. The block placer uses an array of *planes* to keep track of the blocks of each core type available under the current placement. Individual device resources are marked as used or free on a common *use grid*. Note that the placement of a single block, here a block of type *core_0*, typically conflicts with multiple other blocks on each plane.

placed, and moving it to a new randomly selected placement, displacing or evicting any other instances whose placements are newly in conflict.

The primary challenge of adapting this process to place blocks, rather than atomic device resources, is the need for efficient bookkeeping. For example, to determine if a particular block is in conflict with other placed blocks, it is necessary to check if any of the device resources under its footprint are already used by the placement of another instance, which is equivalent to checking if the block footprint intersects the footprint of any other instance. Because this check is performed every iteration, it is also essential that the check be performed efficiently.

In BPR's placer, we require two pieces of information in each iteration: which blocks of each core type are in conflict, and which instances, through which block placements, are making use of each device resource. BPR efficiently determines this information through a data structure resembling Figure 6. We maintain a grid representing the device cell layout (e.g. LAB or M9K) as a common reference for area utilization. Each coordinate in this grid contains two pointers, referencing the instance and block (a *placement*) making use of the cell, or *nil* if the resource is free. Upon choosing a random placement during an iteration, this grid can be used to efficiently find any affected instances for displacement or eviction. We track the availability of placement blocks through an array of *planes*, one for each core type used in the netlist. Each plane holds a grid of legal placements, referenced by origin, as well as *used* and *unused* lists, permitting efficient random access to available placements (e.g. for displacement operations).

One other challenge faced by block placement is that, in terms of block performance, all placements are not created equal. Because the distribution of logic and, to a greater extent, the distribution and connectivity of routing resources varies across the device, there is also variation in the f_{max} achieved for the same core placed at different locations. The amount of variation depends on the core, but can be significant in the worst cases, with deviations of up to 20% for large blocks such as memory interfaces. The BPR placer handles this issue by incorporating the frequency of the minimum-performing block in the current solution as a term in the cost function, preferring the selection of faster blocks in these situations.

3.4 Inter-block Routing

Once the block placer has determined the location of each block instance, and thus the specific intra-block PAR being used, it is the inter-block router's task to stitch the design together by routing the nets connecting each placed block. Inter-block routing is identical to traditional FPGA routing, except that much of the design has already been routed as part of the placer's selection of blocks, and, in this sense, the router has less flexibility to minimize net delays. Due to these similarities, we chose the PathFinder [8] negotiated-congestion router as the basis for the inter-block router in BPR.

The router begins by loading the device's routing resource graph, stored alongside cores in the core database. Because of the large size of modern FPGAs, one of the major challenges in making BPR's router fast was finding an efficient in-memory representation for this graph. Our representation stores the graph as an array of grids, one for each permutation of resources (e.g. R4-C4 or R4-R4), using the device's coordinate system. For each resource pair, the connectivity at each coordinate, equivalent to an individual switch- or connection-box, is stored efficiently as a grid of bitfields for offsets in a neighborhood around the driving resource's (x, y) position. In the database, space is saved by sorting these connectivity patterns and storing each unique pattern, or box topology, exactly once. The router's dynamic cost data is stored similarly. In addition to providing an efficient representation of the device's resource graph, this method of storage has other advantages when performing pathfinding, which will be discussed later.

After loading the device's routing graph, the router loads the routing resource utilization data from the core databse for each block appearing in the final placement. This data represents routes that BPR's router cannot disrupt. To ensure that BPR will not overuse these routing resources, this data is used to break connections in the router's representation of the device's routing resource graph. The endpoint routing resources corresponding to blocks' ports are also loaded at this time and used to annotate the nets.

With the newly modified graph, BPR routes all nets using essentially the same process described in PathFinder [8]. In the details, though, we have made a couple of optimizations to keep run times low. First, instead of using Dijkstra's method to find shortest-paths, we have used a much faster implementation of A* [24], providing speedups of up to 16x on large devices. Second, we reduce the effective size of the routing resource graph when exploring shortest-paths by making the approximation that wire cost is independent of a wire's index within the channel and depends only on the distance the wire covers. Under this assumption, our router simultaneously considers all legal wire indexes at each point in the search, leveraging our bitfield implementation for the storage of resource connectivity to track this information efficiently as an accumulation. The search can progress from

a vertex as long as some index is valid in this accumulation. When the search terminates, the exact index used at each step is determined through backtracking, selecting arbitrarily when multiple paths are possible.

4. EXPERIMENTS

In this section we evaluate the performance of BPR compared to the traditional full-detail FPGA placement and routing flow implemented in commercial tools. We evaluate BPR's performance for a number of case study netlists incorporating a variety of cores, implemented in BPR's custom netlist format. For each case study, we compare BPR's execution time against the time required for Altera Quartus to implement the same circuit from an equivalent VHDL design using the same core HDL, treating synthesis and place and route times separately. We also evaluate the performance overhead of BPR's methodology by comparing the area utilization and maximum clock rate f_{max} achieved by each tool.

In our analysis, we compare against Altera Quartus 9.1 SP2 using low placement effort and optimization level settings for faster compilation at the expense of some circuit quality. Execution times were compared on a quad-core 2.66 GHz Intel Xeon W3520 workstation with 12GB RAM, running CentOS 6.1 x86_64. We compare circuit implementations for the Altera Cyclone III family of devices [21].

4.1 Case Studies

We evaluated BPR's performance on 13 case studies, shown in Table 1, covering a variety of applications. *SAD* computes the sum-of-absolute differences for image comparison. *Gaussian* performs a Gaussian blur on an image. *Sobel* performs image edge detection. *FIR* is a finite impulse response filter. Where appropriate, we tested multiple instances of the same netlist at different sizes (e.g. by changing the kernel size for the image processing applications) to demonstrate how BPR's performance scales with problem size. In many cases we also evaluate both single-precision floating point (indicated by *float*) and 16-bit fixed point versions of the same circuit. To determine the effect of device size, we tested BPR with two devices in the Cyclone III device family: the small EP3C5 (EP3C5F256C8) with 5k logic elements, 46 M9K memories, and 23 DSPs; and the large EP3C120 (EP3C120F484C8) with 119k logic elements, 432 M9Ks, and 288 DSPs.

Most of the case studies in Table 1 incorporate large fixed-point or floating-point arithmetic datapaths. The arithmetic operators in these datapaths consist of tens (fixed point) to hundreds (floating point) of LUTs and make use of the Cyclone III architecture's DSP and M9K memory units when applicable, as determined by the vendor tools' synthesis optimizations. The floating-point operators were generated from Altera Megafunctions [20]. Because arithmetic components are prevalent in common applications of FPGAs, and are often instantiated in large quantities within a single netlist, they were added to the database without sifting of low-valued placements to maximize flexibility, with database population requiring up to several hours running on one workstation.

The 2D DSP examples (e.g. *Gaussian 3x3 1080p 16b*) also include sliding window interfaces to external memory, with the various window and image dimensions specified for each instance through corespec parameters wrapping a common parameterized VHDL implementation. Because these memory interfaces are large and appear in limited quantities in most applications, high sifting values were used to speedup database population by discarding all but a small percentage of possible placements, limiting the collective time required for database population to under an hour. Although this sifting limits placement flexibility for instances of these memory interfaces, we have found that good results can still be achieved for designs whose other components (e.g. arithmetic operators) retain high placement flexibility. Note that these decisions were made on an ad-hoc basis, and are likely not optimal. We leave a detailed analysis of sifting tradeoffs for different core types and sizes as future work.

4.2 Results

Table 1 illustrates BPR's place and route speedup, device resource utilization, and performance overhead for each case study. The left-most major column gives a short description of the case study application as well as the Cyclone III device it was compiled for. The second major column compares BPR's execution time to the time required for Quartus' PAR flow (`quartus_fit`) to place and route an equivalent VHDL design. Because the original HDL source for each core is retained in the core database, the BPR tool is capable of generating these equivalent designs itself, ensuring identical core HDL is used for a fair comparison.

For BPR, we measure execution time including the time to initialize BPR's PAR data structures with the mapped netlist, and ending when the final PAR solution is produced in memory. We exclude the time required to export the solution to Quartus in the large human-readable QSF and RCF formats. The PAR time reported for Quartus is the elapsed time reported by `quartus_fit` when run with low effort settings for placement and physical optimizations. Note that Quartus also does not write out QSF and RCF constraints in the normal course of PAR, requiring the use of a second tool (`quartus_cdb`) when back-annotating this data. We exclude the time required to run `quartus_cdb`, which is typically on the order of a few seconds, in the times for Quartus.

The results show an average 93x PAR speedup over all case studies, requiring an average of 1.1s. The speedup numbers are higher on average for the floating-point examples, at 154x, compared to the fixed-point examples, at 76x. This larger speedup is caused by a larger reduction in problem size for the floating-point examples over detailed PAR, with a typical float operator containing 100s of LUTs, compared to 10s for the fixed-point operators, all of which is hidden behind black-box footprints in BPR.

The third major column shows the percentage of device resources required to implement the design as reported by Quartus for each netlist case study, without BPR's PAR constraints. We include this data to give an idea of the size of each design, which impacts the execution times of placement (by complicating fit) and routing (through increased obstacles). Note that although area overhead due to overly generic cores is possible in BPR's methodology, it was not significant for these case studies. *LABs* are an Altera primitive consisting of 16 logic elements, *LEs*, which each consist of a configurable LUT. *DSP* corresponds to multiplier blocks and *M9K* represents on-chip block RAMs.

BPR was able to use up to 49% of the small (C5) Cyclone III when working with fixed-point datapaths. This high utilization is possible because of the placement flexibil-

Table 1: BPR place and route speedup and overhead of case study circuits.

Applications		Place and Route (PAR) Times					Area				Clock		
Netlist	Device	BPR	BPR Placer	BPR Router	Quartus	Speedup	LABs	LEs	DSP	M9K	BPR	Quartus	Overhead
SAD 3x3 480p 16b	C5	0.629	0.039s	0.59s	0m 11s	**17x**	45%	38%	0%	13%	72 MHz	73 MHz	**1%**
FIR 14-tap 16b	C5	0.076s	0.048s	0.028s	0m 9s	**118x**	56%	34%	61%	0%	222 MHz	225 MHz	**1%**
FIR 20-tap 16b	C5	0.176s	0.12s	0.056s	0m 13s	**74x**	80%	49%	87%	0%	208 MHz	225 MHz	**8%**
Gaussian 3x3 1080p 16b	C120	0.285s	0.239s	0.046s	0m 38s	**133x**	6%	2%	3%	3%	69 MHz	76 MHz	**10%**
Gaussian 5x5 1080p 16b	C120	0.595s	0.478s	0.117s	0m 53s	**89x**	15%	4%	9%	5%	63 MHz	75 MHz	**19%**
Gaussian 7x7 480p 16b	C120	2.059s	1.772s	0.287s	1m 9s	**34x**	31%	8%	11%	4%	62 MHz	64 MHz	**3%**
FIR 48-tap 16b	C120	1.319s	1.168s	0.151s	0m 54s	**41x**	24%	5%	17%	0%	157 MHz	224 MHz	**43%**
FIR 76-tap 16b	C120	2.857s	2.672s	0.185s	1m 33s	**33x**	38%	8%	26%	0%	124 MHz	224 MHz	**81%**
FIR 96-tap 16b	C120	3.69s	3.473s	0.217s	1m 35s	**26x**	48%	10%	33%	0%	95 MHz	224 MHz	**137%**
Sobel 1080p 16b	C120	0.188s	0.156s	0.032s	0m 35s	**186x**	2%	2%	4%	0%	62 MHz	77 MHz	**24%**
SAD 3x3 1080p float	C120	0.768s	0.662s	0.106s	1m 58s	**154x**	23%	17%	0%	11%	51 MHz	72 MHz	**40%**
FIR 9-tap float	C120	0.41s	0.354s	0.056s	1m 18s	**190x**	16%	11%	11%	2%	123 MHz	140 MHz	**14%**
FIR 16-tap float	C120	1.171s	1.051s	0.12s	2m 19s	**119x**	29%	21%	19%	3%	109 MHz	146 MHz	**34%**
Average		1.094s			1m 2s	93x	32%	16%	22%	3%	109 MHz	142 MHz	32%
Geometric Mean		0.637s			0m 46s	72x	23%	10%			97 MHz	124 MHz	14%

ity of the smaller fixed-point cores, one of which (add16) is purely logic. These smaller cores also achieve high packing ratios due to low internal communication requirements, as discussed in Section 3.2.3, resulting in a higher ceiling for area. Higher utilization was also achieved for the floating-point DSP circuits on the large (C120) device. However, for these larger cores, another factor limits circuit size: competition for the device's scarce resources such as DSPs and M9Ks. The typical floating-point core incorporates one or more of these resources, but has a black-box footprint (due to logic requirements) that covers many more, leading to a large amount of waste. We plan to address this challenge as future work, and discuss this problem in more detail in that section.

The right-most major column compares the maximum frequency f_{max} of BPR's exported designs against designs using Quartus' full-detail place and route. For each case study, we report the clock frequency achieved by BPR's place and route against the frequency achieved by Quartus from the equivalent HDL design without constraints. The frequency for BPR is the value reported by Quartus timing analysis after running BPR's constrained design through `quartus_fit` for finalization, as discussed in Section 3.1. In all cases we report the unrestrained f_{max} number reported by `quartus_sta`.

BPR's speedups were achieved with an average 34% reduction in circuit frequency. This overhead was lowest for the small C5, down to 1%, in part because of the reduced impact of poor quality placements on a small device. The highest overhead numbers were experienced by the large fixed-point FIR filters on the C120, at up to 137%. Part of the large overhead of these large circuits is due to a limitation in BPR's current router where it doesn't have access to very long wires in the Cyclone III architecture (e.g. C16). Note that although the worst-case overhead for these circuits is significant, they are comparable to previous work with typical 2-4x overhead [19]. For many of the 2D DSP examples, e.g. the *Gaussian* kernels, the circuit's performance overhead is determined by the f_{max} of the sliding-window buffer. Here, the overhead is caused by the need to confine the window core to a compact rectangular footprint, as discussed in Section 3.2.3. These examples illustrate that area/frequency tradeoffs are especially important for slower cores which may set a circuit's f_{max} before other, faster components or inter-block routes. We leave a thorough evaluation of these tradeoffs as future work.

5. FUTURE WORK

The performance of BPR's placer is often limited by crowding around special device resources such as DSPs and M9Ks, which are scarce compared to LABs and are arranged densely in columns on the Cyclone III. It is common for a single core (e.g. floating-point arithmetic operators) to incorporate only one or two such resources along with a large amount of logic, which has the effect of requiring large block footprints. Because BPR is limited to placing these cores as rectangular blocks which cannot overlap, a single block will often cover multiple unused DSPs and M9Ks, preventing access to these resources by other blocks. We plan to address this in future versions of BPR by stripping out such scarce resource types when they're used by a core, to be placed and routed separately as small sub-blocks.

The limitations of BPR's current placer are exacerbated by BPR's simplistic core mapper. In the current version of BPR, core mapping does not consider the limited number of resources such as DSPs available on the device. Thus, the core mapper will continue to map core instances to implementations making use of DSPs and other scarce resources even after all such resources on the device have been exhausted, making placement of subsequently mapped blocks impossible. We plan to address this in future versions of BPR by expanding core database population to include secondary core implementations that don't include scarce resources, giving the mapper other options in these situations.

The performance of BPR's router is also limited by the inflexibility of the many intra-block routes imported after placement, which are currently immutable. In addition to reducing the f_{max} achieved by BPR, this inflexibility can potentially make large designs with dense blocks unroutable. We plan to extend future versions of BPR by allowing the router to rip-up imported intra-block routes as necessary in these situations, possibly subject to some tunable effort level to allow for tradeoffs between quality and execution time.

6. CONCLUSIONS

In this paper, we introduced BPR, a new FPGA CAD tool for the fast compilation of FPGA circuits that exploits the functional reuse common in modern FPGA design. BPR modifies traditional FPGA place and route algorithms to abstract the low-level implementation details of reused cores in a design, with the corresponding reduction in problem size enabling large speedups over full-detail commercial FPGA

place and route, while also requiring significantly less memory. The main limitation of BPR is performance (f_{max}) overhead and limited scalability due to reductions in placement and routing flexibility caused by limiting the implementation of cores to immutable rectangular footprints. However, we show that over 13 case studies covering a variety of applications, the current version of BPR achieves an average 93x speedup, with a relatively low 34% average reduction in circuit f_{max} compared to full-detail PAR performed by commercial tools, which is acceptable for many applications. Furthermore, these results bring FPGA compilation times much closer to those of competing technologies including GPUs, potentially enabling FPGA support for emerging just-in-time compilation models such as OpenCL. Future work addressing limitations in the current version of BPR, identified in this paper, may make BPR and similar approaches more competitive with traditional FPGA compilation flows.

7. ACKNOWLEDGMENTS

This work was supported in part by the I/UCRC Program of the National Science Foundation under Grant Nos. EEC-0642422 and IIP-1161022. The authors gratefully acknowledge vendor equipment and/or tools provided by Altera and Synopsys.

8. REFERENCES

[1] A. DeHon, "The Density Advantage of Configurable Computing," *Computer*, vol. 33, pp. 41–49, April 2000.

[2] G. Stitt and F. Vahid, "Energy Advantages of Microprocessor Platforms with On-Chip Configurable Logic," *IEEE Des. Test*, vol. 19, pp. 36–43, November 2002.

[3] E. Girczyc and S. Carlson, "Increasing design quality and engineering productivity through design reuse," in *Proceedings of the 30th international Design Automation Conference*, ser. DAC '93, 1993.

[4] 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*, ser. FPGA '01, 2001.

[5] *SOPC Builder User Guide*, Altera Corporation, December 2010.

[6] OpenCores. (2012, Mar.) OpenCores. [Online]. Available: http://opencores.org/projects

[7] 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*. London, UK: Springer-Verlag, 1997, pp. 213–222.

[8] 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, 1995.

[9] J. Becker, T. Pionteck, C. Habermann, and M. Glesner, "Design and Implementation of a Coarse-Grained Dynamically Reconfigurable Hardware Architecture," in *Proceedings of the IEEE Computer Society Workshop on VLSI 2001*, ser. WVLSI '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 41–.

[10] G. K. Rauwerda, P. M. Heysters, and G. J. M. Smit, "Towards software defined radios using coarse-grained reconfigurable hardware," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 16, pp. 3–13, January 2008.

[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, 2010.

[12] S. Shukla, N. W. Bergmann, and J. Becker, "QUKU: A Two-Level Reconfigurable Architecture," in *Proceedings of the IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 109–.

[13] A. Koch, "Structured design implementation: a strategy for implementing regular datapaths on FPGAs," in *Proceedings of the 1996 ACM fourth international symposium on Field-programmable gate arrays*, ser. FPGA '96, 1996.

[14] R. Lysecky, F. Vahid, and S. X. D. Tan, "A Study of the Scalability of On-Chip Routing for Just-in-Time FPGA Compilation," in *FCCM '05: Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 57–62.

[15] R. Lysecky, F. Vahid, and S. X.-D. Tan, "Dynamic FPGA routing for just-in-time FPGA compilation," in *DAC '04: Proceedings of the 41st Annual Conference on Design Automation*. New York, NY, USA: ACM, 2004, pp. 954–959.

[16] C. Mulpuri and S. Hauck, "Runtime and quality tradeoffs in FPGA placement and routing," in *FPGA '01: Proceedings of the 2001 ACM/SIGDA Ninth International Symposium on Field Programmable Gate Arrays*. New York, NY, USA: ACM, 2001, pp. 29–36.

[17] P. Athanas, J. Bowen, T. Dunham, C. Patterson, J. Rice, M. Shelburne, J. Suris, M. Bucciero, and J. Graf, "Wires on demand: Run-time communication synthesis for reconfigurable computing," in *FPL '07: International Conference on Field Programmable Logic and Applications*, Aug. 2007, pp. 513–516.

[18] R. Tessier, "Frontier: A Fast Placement System for FPGAs," in *VLSI: Systems on a Chip, IFIP TC10/WG10.5 Tenth International Conference on Very Large Scale Integration (VLSI 99), December 1-4, 1999, Lisbon, Portugal*, ser. IFIP Conference Proceedings, L. M. Silveira, S. Devadas, and R. A. da Luz Reis, Eds., vol. 162. Kluwer, 1999, pp. 125–136.

[19] C. Lavin, M. Padilla, J. Lamprecht, P. Lundrigan, B. Nelson, and B. Hutchings, "HMFlow: Accelerating FPGA Compilation with Hard Macros for Rapid Prototyping," in *Proceedings of the 2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*, ser. FCCM '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 117–124.

[20] *Quartus II Handbook Version 11.1*, Altera Corporation, November 2011.

[21] *Cyclone III Device Handbook*, Altera Corporation, December 2011.

[22] *Application Note 161: Using the LogicLock Methodology in the Quartus II Design Software*, Altera Corporation, June 2003.

[23] V. Betz and J. Rose, "FPGA routing architecture: segmentation and buffering to optimize speed and density," in *Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*, ser. FPGA '99, 1999.

[24] R. Dechter and J. Pearl, "Generalized best-first search strategies and the optimality of A*," *J. ACM*, vol. 32, no. 3, pp. 505–536, Jul. 1985.