Intermediate Fabrics: Virtual Architectures for Circuit Portability and Fast Placement and Routing

James Coole, Dr. Greg Stitt
University of Florida
Department of Electrical & Computer Engineering
Gainesville, FL, USA

jcoole@ufl.edu, gstitt@ece.ufl.edu

ABSTRACT

Although hardware/software partitioning of embedded applications onto FPGAs is widely known to have performance and power advantages, FPGA usage has been typically limited to hardware experts, due largely to several problems: 1) difficulty of integrating hardware design tools into well-established software tool flows, 2) increasingly lengthy FPGA design iterations due to placement and routing, and 3) a lack of portability and interoperability resulting from device/platform-specific tools and bitfiles. In this paper, we directly address the last two problems by introducing *intermediate fabrics*, which are virtual reconfigurable architectures specialized for different application domains, implemented on top of commercial-off-the-shelf devices. Such specialization enables near-instantaneous placement and routing by hiding the complexity of fine-grained physical devices, while also enabling circuit portability across all devices that implement the intermediate fabric. When combined with existing work on runtime synthesis from software binaries, intermediate fabrics reduce the effects of all three problems by enabling transparent usage of COTS FPGAs by software designers. In this paper, we explore intermediate fabric architectures using specialization techniques to minimize area and performance overhead of the virtual fabric while maximizing routability and speedup of placement and routing. We present results showing an average placement and routing speedup of 554x, with an average area overhead of 10% and clock overhead of 18%, which corresponds to an average frequency of 195 MHz.

Categories and Subject Descriptors
J.6 [**Computer-Aided Enginering**]: Computer-aided Design

General Terms
Performance, Design

Keywords
intermediate fabrics, placement and routing, virtualization, FPGA, speedup

1. INTRODUCTION

Partitioning embedded applications onto field-programmable gate arrays (FPGAs) has been widely shown to have significant performance [9] and power [33] advantages over software-only execution. Despite these advantages, FPGA usage has been limited due to increased application design complexity largely resulting from three main problems: increasingly long placement and routing times, a lack of circuit portability, and difficulty of integrating circuit design tools into software tool flows.

Increasingly long execution times for placement and routing (PAR) is an emerging problem that can require hours, days [4], and even more than a week [29] for very large circuits. FPGA PAR execution times thus represent a significant design bottleneck, which consequently complicates debugging and verification, reduces productivity, increases nonrecurring engineering costs, and increases time to market. Furthermore, long PAR times are a barrier to more mainstream FPGA usage [25][28][29], where well-established methodologies rely on rapid compilation times.

Widespread FPGA usage has also been limited by the lack of circuit portability, even across devices in the same family. Despite portability being an important factor in the acceptance of popular microprocessors, few studies have focused on establishing portability for FPGAs. As a result, redesigning circuits for different devices is often time consuming and costly, especially when those circuits use device-specific cores. Lack of portability further complicates design productivity by preventing third-party design tools from supporting specific devices and platforms [17][26].

An additional problem preventing more mainstream FPGA usage is the difficulty of integrating circuit-design tools into software tool flows. Although high-level synthesis tools have been introduced to provide C-like syntax, software designers have been very reluctant to change languages or to change well-established compilers, debuggers, and development environments [35]. Previous work has focused on hiding the FPGA by dynamically

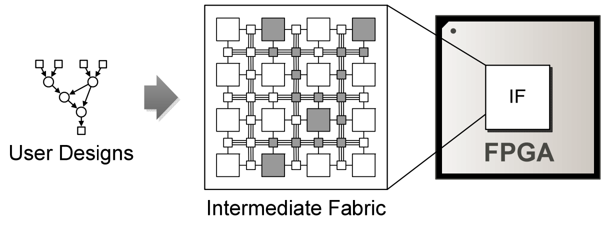

Figure 1: *Intermediate fabrics* (IFs) are virtual fabrics implemented on FPGAs that enable portability of netlists across devices and fast placement and routing of netlists**.**

synthesizing circuits from software binaries [2][35], but those approaches require specialized FPGAs instead of commercial-off-the-shelf (COTS) devices.

To address these problems, we introduce *intermediate fabrics* (IFs). As shown in Figure 1, intermediate fabrics are virtual reconfigurable fabrics specialized for different application domains, implemented between user designs and the underlying physical FPGA (i.e., an *intermediate* translation layer). From the point of view of the application designer, an IF looks like any other reconfigurable device that is programmed using a configuration bitstream.

However, unlike a physical device, whose architecture must support a wide range of applications, IFs can be specialized for particular application domains or even individual applications. Such specialization hides much of the complexity of fine-grained COTS devices, thus enabling fast placement and routing. Although it is widely known that coarse-grained fabrics have reduced PAR times, intermediate fabric enable such speedups on fine-grained COTS devices. In addition, because intermediate fabrics are virtual devices, they enable portability across any physical device that can implement the fabric.

When combined with existing high-level synthesis or runtime synthesis techniques [2][35], IFs enable more transparent usage of COTS FPGAs by mainstream designers. In addition, IFs potentially enable mainstream designers to view FPGAs largely in the same way as other accelerator technologies with rapid compilation times, such as graphics processing units (GPUs), by using languages such as OpenCL. Other advantages include partial reconfiguration on devices that lack architectural support, abstraction of multiple devices (e.g., one large IF spanning multiple FPGAs), and physical device transparency for long life-cycle applications, where a virtual fabric can hide changes in underlying devices that may occur due to changes in supply chains over time.

The main limitation of IFs is area overhead incurred by a virtual fabric. Although the current area overhead prevents IFs from being used for very large circuits used in state-of-the-art FPGAs, we show that IFs can efficiently implement common accelerator kernels, while achieving near-instantaneous placement and routing times and circuit portability across devices. We evaluate intermediate fabrics with sizes up to 96 DSP units (which is the amount of DSP48 units available on Virtex 4 LX devices) and show that IFs implemented on a Virtex 4 LX200 can support up to 225 DSP units, which enables circuits larger than those used by previous studies involving ASIPs [18], coarse-grained architectures [32], high-level synthesis tools [15], and dynamic synthesis tools [35], therefore making IFs a complementary technology to those studies. Note that IFs are not intended to replace physical design tools for FPGAs and are instead intended to support FPGA usage models where scalability and overhead are not critical issues. For example, IFs could be used by ASIPs to define custom instructions, or as the target of high-level synthesis tools that create co-processors for software acceleration. On average, the IFs evaluated in this paper achieved a PAR speedup of 554x, with an average area overhead of 10%, clock overhead of 18%, and frequency of 195 MHz.

In this paper, we make the following contributions:

- Establishment of the feasibility of using virtual coarse-grained reconfigurable fabrics on COTS FPGAs, while achieving near-instant placement and routing and portability across devices.
- Introduction of a family of intermediate fabrics for data-parallel circuits that achieve enough scalablity to enable realistic usage scenarios (e.g., up to 225 DSP units).
- Exploration of specialization techniques and architectural tradeoffs for data-parallel circuits to minimize IF overhead while maximizing PAR speedup.
- Determination of Pareto optimal IFs in terms of overhead and routability for both application-specialized fabrics and IFs intended for general purpose usage.

2. PREVIOUS WORK

Numerous specialized reconfigurable architectures have been proposed for individual domains [3][11][23]. These architectures are motivating examples for intermediate fabrics, which potentially enable similar improvements on COTS devices and avoid the high cost of custom devices.

Totem [7] investigated automatically generating custom coarse-grained reconfigurable architectures for specific domains (e.g, RaPiD for DSP [11]). Hammerquist presents a similar approach for application-specific FPGAs [16]. Both these studies have similar goals as IFs, but focus on integrating custom fabrics into ASIC devices. IFs aim to virtualize custom reconfigurable architectures to enable usage on COTS devices. Exploration and customization techniques from both approaches are complementary and could be used to create custom IF structures. In addition, architecture-adaptive PAR techniques [31] are also complementary and could be used to enable more effective use of novel IF fabrics.

Previous studies have also focused on custom placement techniques for coarse-grained datapaths [6][20] to avoid added complexity resulting from conversion to fine-grained FPGA components. Although such approaches are good technical solutions, they are limited to datapath synthesis and achieve PAR speedup ranging from 3.2x to 4.5x [6]. IFs achieve an average PAR speedup of 554x, and are therefore an effective complementary alternative for situations where area overhead is not critical. An additional practical advantage of IFs is that by hiding the physical device, the PAR tools do not require knowledge of proprietary low-level architecture details, and can be thus be used on potentially any COTS device.

Previous work has also investigated FPGA overlay networks [19] to provide specialized virtual networks (e.g., time-multiplexed) to more efficiently support highly-interconnected circuits where each connection has low bandwidth requirements. IFs are complementary and could potentially integrate overlay networks into the virtual fabric architecture.

Warp processors [35] originally addressed lengthy PAR times by using a PAR-specialized fabric with on-chip CAD tools to enable runtime synthesis and PAR. Similarly, Beck [2] introduced a processor that dynamically translated Java bytecode onto a coarse-grained reconfigurable fabric. Although those approaches achieved order of magnitude PAR speedup, all CAD was limited to a specialized device. IFs remove this limitation by enabling fast PAR for COTS devices. In fact, IFs are complementary, potentially enabling warp processing or similar techniques on COTS devices.

Although PAR is a widely studied problem, most approaches have focused on improving routability or timing [5][13][24], leaving PAR execution time as a secondary consideration. Lysecky introduced dynamic FPGA routing [22] and JIT FPGA compilation [21] to perform fast placement and routing, but that work also assumed a specialized FPGA fabric. Mulpuri and Hauck [27] studied tradeoffs between routing quality and execution time, showing that a 3x PAR speedup can be achieved with a 27% degradation of circuit performance. Although a 3x speedup is significant, increasingly long PAR times require a larger reduction to increase designer productivity. Wires on Demand [1] introduced a fast PAR technique used for partial reconfiguration of communication between pre-placed and routed modules. IFs are complementary and could potentially be combined with Wires on Demand, using IFs for module PAR and Wires on Demand for communication between modules.

Previous work has also focused on virtual reconfigurable architectures [30][36] that enable dynamic partial reconfiguration and evolvable hardware. Quku [32] is a coarse-grained array of ALUs, implemented on top of an FPGA, which can be rapidly reprogrammed using a soft core microprocessor. Although conceptually similar, IFs also address problems of PAR execution time, circuit portability, and long-life cycle applications. Quku is essentially one instance of an IF.

3. INTERMEDIATE FABRICS

This section describes the architecture for an IF family targeting data-parallel applications (Section 3.1), the IF tool flow (Section 3.2), and usage models and scenarios (Section 3.3).

3.1 Architecture

IFs can potentially implement any fabric architecture. Therefore, it is outside the scope of this paper to discuss the near-infinite possibilities. In this paper, we limit our exploration of IFs to a family of IFs intended for data-parallel applications, which are commonly implemented as pipelined circuits on FPGAs. Note that we present the fabric abstractly to avoid suggesting a specific fixed architecture, as the fabric can be specialized in numerous ways. The experiments in Section 5 present specific examples.

As opposed to COTS FPGAs, which generally have a somewhat uniform fabric, IFs may be decomposed into different 'planes' that are specific to different aspects of a circuit (e.g., control, data), although implementations of multiple planes need not be mutually exclusive. As shown in Figure 2 for the data-parallel IF

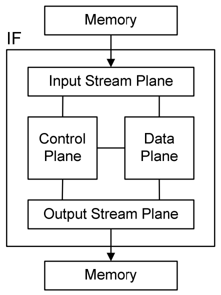

Figure 2: An example of a data-parallel IF that is decomposed into separate coordinating planes, each with an architecture customized to its specific function.

family discussed in this paper, the architecture consists of three types of planes: *data*, *control*, and *stream*.

Note that the presented architecture is not intended to be general purpose, and may not provide good support for implementing arbitrary netlists. In Section 5.3, we evaluate the possibility of using these IFs as general purpose fabrics, however, we expect that a user or design tool would instead select an appropriate specialized fabric when implementing a particular netlist. Usage scenarios are discussed in more detail in Section 3.3.

3.1.1 Data Plane Architecture

The data plane of IFs used in this paper consists of a traditional, island-style topology, with *computational units* (CUs) distributed across the fabric in a grid, with routing resources (e.g., tracks, connection boxes, switch boxes) filling the space in between, as illustrated in Figure 3. The basic structure of the data plane is identical to traditional island-style FPGA fabrics, except that the plane is virtual and has each component (CUs, tracks, connection boxes, and switch boxes) specialized for different applications or domains. IFs may replicate this island-style structure an arbitrary number of times to form any size with any aspect ratio.

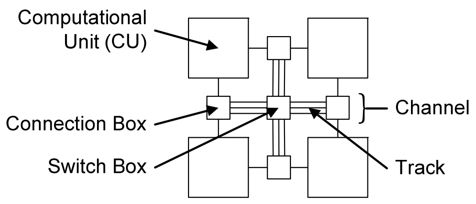

Figure 3: Data plane architectural components.

Computational units (CUs) provide resources for the mathematical or logical operations in design netlists, and are analogous to the CLBs or DSP units in FPGAs. The function performed by the CUs varies between fabrics with different specializations, and a fabric might contain multiple types of CUs. DSP-specialized IFs, for example, might contain a mix of multipliers and adders, whereas an IF specialized for scientific computation might contain ALUs capable of performing a variety of arithmetic and logical operations, depending on their configuration. As a concrete example, the majority of CUs used in this paper are mapped directly onto Xilinx DSP48 units, with some additional shift registers to handle realignment for pipelined routing, as discussed in Section 3.2. Specific fabric architectures are discussed in Section 5, where we also evaluate floating-point CUs.

Tracks are the fundamental IF routing resources. However, unlike single-wire tracks in FPGAs, IF tracks can be multiple wires wide. For example, 16-bit tracks can be provided to connect to CUs with 16-bit outputs. Connection boxes connect CU inputs and outputs to routing tracks in adjacent channels. IFs can specialize both the number and flexibility of connection boxes depending on the routing requirements of a particular domain or application. Similarly, switch boxes connect tracks to other tracks in intersecting channels or to distant tracks in the same channel. Although IFs can potentially specialize the topology of switch boxes, in this paper all switch boxes use a planar topology.

Unlike physical devices, IF I/O can be placed anywhere in the fabric, which could potentially reduce routing requirements for certain applications. However, all fabrics evaluated in this paper contain I/O on the periphery of the fabric.

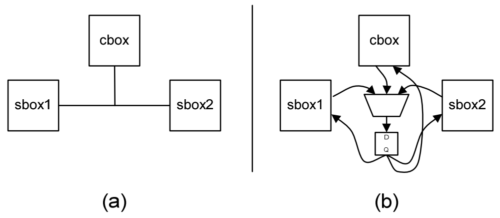

Figure 4: Intermediate fabric RTL implements (a) virtual routing tracks using (b) a MUX to select from each possible track source and a register to prevent combinational loops.

Although not shown in the figures, the data plane contains configuration flip flops that are chained together to form a long shift register that programs the plane serially.

We map the virtual data plane onto a physical FPGA as follows. The entire fabric is represented as a structural VHDL entity that instantiates all IF components (CUs, tracks, switch boxes, connection boxes) and connects them together. Each type of IF component is defined using a separate VHDL entity. For each type of CU, we use a VHDL entity that implements the corresponding behavior. For example, a multiplier CU is implemented using a multiplier entity, which the FPGA device tools map onto a DSP unit. For each connection/switch box, we use MUXs to implement each possible connection.

IF tracks are mapped onto physical resources as shown in Figure 4. Virtualizing bidirectional tracks in the IF requires using MUXs from all possible sources to a single sink. The IF PAR tools determine the appropriate source for each track and create a bitstream that selects the correct MUX output using a configuration register that is set when the fabric is configured. Due to the need for MUXs, IF tracks have the potential for a significant area overhead. For example, an *m*-bit track with *n* possible sources is mapped to *n*+1 busses and a *m*:1 *n*-bit MUX. As the example in Figure 4 shows, a 16-bit wide track connected to 3 possible sources requires at least 64 wires (4 16-bit busses + fanout) and a 3:1 16-bit MUX.

The data plane uses pipelined routing resources to maximize performance and eliminate the possibility of combinational loops in the resulting HDL. To avoid computationally-expensive PAR algorithms for pipelined interconnect [12], the data plane includes variable shift registers on CU inputs to realign pipeline stages in a way similar to [34]. While such an architecture is not appropriate for general netlists, we have observed that the pipelined interconnect often works well for highly data-parallel circuits.

As future work, we plan to directly map virtual routing tracks onto physical routing tracks, which has the potential to greatly reduce the IF overhead. However, the current approach has the advantage of being device and vendor independent as well as being easily integrated into existing RTL designs and tool flows.

3.1.1 Control Plane Architecture

The control plane provides basic primitives to implement Moore state machines and control logic: a state register, next state logic, state-dependent output logic, and state-independent output logic. We implemented the control plane architecture on FPGAs using two LUTs: one with synchronous reads implemented on block RAM and one with asynchronous reads implemented using distributed RAM.

The synchronous LUT implements the state register, next state logic, and state-independent output logic by using an address that corresponds to the current state and the current inputs. For every state and input combination, the synchronous LUT stores the next state and the output values.

The asynchronous LUT implements state-independent output logic (e.g., pipeline stalls due to full buffers in the stream plane) by storing output values for every input combination.

One obvious limitation is that this control plane will not scale to large numbers of control inputs or large state machines, due to an exponential increase in the LUT sizes. However, this limitation is irrelevant for the targeted data-parallel circuits, which often require few control resources. For the applications evaluated in this paper, the control plane required only 1% of the resources on a Xilinx Virtex 4 LX100 and had a maximum clock frequency of 360 MHz. Future work will investigate IF architectures for control-intensive applications.

3.1.2 Stream Plane Architecture

To effectively support data streams, we use a separate plane to deal with transferring data from external memories into the data plane, thus saving data plane resources for actual computation. In the simplest case, the stream plane consists of address generators that take a base address and a transfer size as input, and then read/write the appropriate locations from memory. When implemented on the FPGA, the stream plane consists of a counter, basic control, and a memory controller.

For certain domains, the stream plane may also use specialized buffers to improve memory bandwidth. Image processing IFs, for example, include smart buffers [10][14] in the input stream plane.

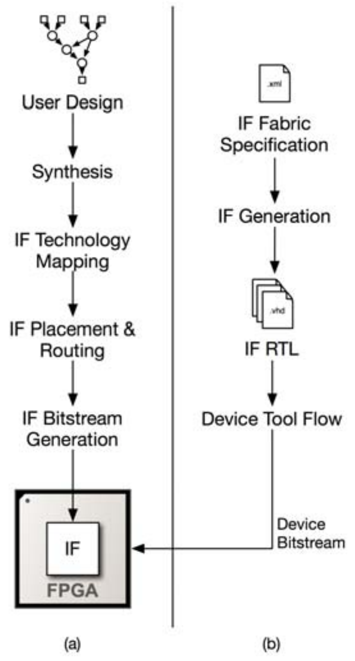

Figure 5: Designers targeting IFs use the (a) IF tool flow to create a configuration bitstream for an IF that has been (b) implemented on a physical FPGA by synthesizing the corresponding IF RTL using the FPGA device tool flow (e.g. Quartus, ISE).

Smart buffers are specialized cache structures that are capable of reusing data read from memory to efficiently generate sliding windows of data that can be passed to the data plane. For example, a smart buffer for Sobel edge detection would generate a 3x3 window of data every cycle for the data plane. Such buffering significantly improves memory bandwidth, which in turn enables more parallelism within the data plane. The image processing IFs used in this paper use smart buffers in the stream plane that are capable of generating a sliding window up to a maximum size of 16x16 32-bit pixels. For all the experiments, the stream plane supports up to 16x16 sliding windows for high-definition images (1920x1080 pixels).

3.2 IF Tool Flow

As shown in Figure 5, intermediate fabrics involve two sets of CAD tools: those specific to the IF (i.e., IF tools in Figure 5(a)), and those specific to the underlying FPGA device (i.e., FPGA device tools in Figure 5(b)). Note that we use "FPGA tools" and "device tools" interchangeably. The FPGA device tool flow is only executed a single time to generate the FPGA bitstream for each IF, at which point all user design modifications use the IF tool flow. Determination of IFs is discussed in Section 3.3.

Because IFs are essentially virtual FPGAs, the IF tool flow is identical to the traditional FPGA CAD tool flow consisting of synthesis (RT or high-level), technology mapping, placement, and routing. However, because IFs will often be specialized to a particular domain, each step of the IF tool flow can also be specialized. Although IF synthesis and technology mapping would be done using existing techniques, we currently perform these two steps manually.

Note that although the fabric uses pipelined interconnect, the IF tool flow avoids use of pipelined PAR algorithms [12] by using shift registers to realign pipeline stages after PAR [34].

IF placement is based on the VPR [5] simulated annealing placement algorithm, but is specialized for the specific IF architectures described in the previous section, which are considerably different than fine-grained FPGA fabrics for which VPR was intended. IF placement modifies the simulated annealing parameters used by VPR (e.g., moves per step, stopping temperature, cooling schedule), by using values which we empirically determined to be a good tradeoff between routability and PAR execution time.

IF routing uses well-known negotiated congestion routing [24]. We specialized IF routing by adjusting the maximum number of routing iterations before assuming that a netlist cannot be routed.

3.3 Envisioned Usage Models

Implementing a design using IFs requires the availability of an appropriate IF. Because systems using IFs will be expected to handle a variety of circuits, those systems will at various times need a variety of different IFs. Currently, we envision two usage models to enable such flexibility. *Note that full realization of these usage models is outside the scope of this paper* and will be the focus of future work.

In the *library model*, the IF tool flow would select an appropriate IF for a particular circuit from a library of pre-made IFs. The library would contain a number of IFs for applications likely to be encountered by the system. IFs inside the library would include an architectural description of the fabric as well as pre-made bitstreams implementing the fabric on a target device. Selection could be manual, or more likely, automatic, based on the resources needed by the circuit and a search of IFs available in the library. Currently, we use a library of several fabrics that contain different CU components as described in the experiments section, with stream planes that vary for image processing and basic streaming. We currently manually select an appropriate IF based on knowledge of the circuit to be implemented.

The primary advantage of the library model is selected IFs are immediately available for implementation on the device, which, when combined with the fast PAR afforded by IFs, results in near instantaneous FPGA implementations. The model also removes dependence on the device vendor tool flow after the library's initial construction. Such an advantage enables third-party tools to more easily target different physical devices. The obvious disadvantages are the space required to store the library, the time required to build the library, and the requirement of a priori knowledge to construct a library sufficient for circuits targeted by the system.

Alternatively, the *synthesis model* would replace or complement the library with a synthesis step that creates a custom IF for a circuit while retaining enough flexibility for similar circuits to reuse the resulting fabric. This model has the potential to produce highly optimized IFs for a particular circuit, but requires a single execution of PAR for the physical device. However, time required for each FPGA PAR is amortized over the lifetime of the IF.

4. SPECIALIZATION TECHNIQUES

In this section, we discuss optimization techniques for specializing an IF to a given application domain such that PAR speedup is maximized and overhead is reduced. For the purpose of comparison between IFs, we define the *IF overhead* as resource utilization that would not be necessary when implementing a circuit directly on the physical device. For example, for an IF whose CUs map directly onto DSP units, the IF overhead would be the number of CLBs used by the fabric, as those CLBs implement the virtual routing resources and configuration register chain. If a circuit was mapped directly to the physical device instead of the IF, the DSP units would still be used, but the CLBs would not be used, and are therefore considered as overhead.

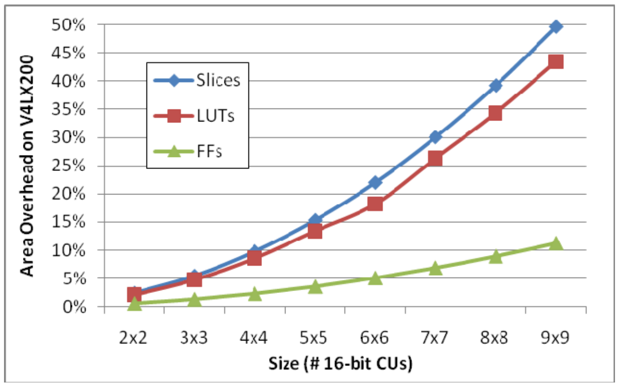

Figure 6: Scalability and area overhead of *unspecialized* IFs for different square fabric sizes (# of 16-bit CUs) with 4 tracks per channel. Although not explicitly shown, virtual routing resources are the main cause of slice and LUT overhead. FF overhead is caused mainly by the IF configuration registers.

Figure 6 summarizes IF area overhead and scalability for unspecialized versions of the IF family described in the previous section, for different sizes of square fabrics with 4 tracks per channel, when implemented on a Xilinx Virtex 4 LX200. For these results, the CUs were 16-bit DSP units and all tracks were 16 bits. Connection boxes were placed on rows only and each box connected only to the inputs of the CU/IO on its south side and the outputs of the CU/IO on its north side – the minimum number and flexibility of connection boxes required to provide all CUs access to the interconnect. These decisions were made to minimize area overhead, and serve as a baseline for the specialization techniques.

Due to space constraints, we omit a breakdown of overhead for each IF component and instead summarize the results. As expected, the high utilization of slices and LUTs is caused by the MUXs used by routing resources, which potentially limits scalability. Motivated by these results, we present a number of specialization techniques to reduce area overhead of routing resources, while minimizing the impact on routability. Note that the techniques aren't necessarily intended to improve both routability and overhead compared to the baseline island-style fabrics. Instead, we implement these techniques to create Pareto optimal fabrics that may trade off area overhead for routability, and vice versa. As shown in Section 5, the overhead in Figure 6 can be greatly reduced without sacrificing much routability. We have implemented IFs that use all 96 DSP48 units on corresponding Virtex 4 LX FPGAs, and for the LX200, we have implemented a 15x15 fabric (225 16-bit DSP CUs). Although scalability does limit IF usage, 225 CUs can support numerous realistic circuits.

Although some of the following techniques have been evaluated for FPGA architectures [4], their effectiveness in minimizing the overhead of a virtual implementation cannot be easily extrapolated from those results. Likewise, the difference between coarse-grain fabrics and fine-grained FPGAs suggests tradeoffs appropriate for IFs need to be evaluated separately. We currently consider five specialization techniques: *track density, long tracks, jump tracks, wide channels,* and *connection box flexibility*.

Track density (i.e., tracks per channel) reduces IF routing resources uniformly in each channel. We found that while the savings in overhead by decreasing the track density is linear, the effect on routability depends on the size of the fabric and the fabric's current routability. The impact on routability is minimized for fabrics with low or high routability and is maximized for fabrics with mid-range routability.

Long tracks skip over switch boxes, which uses fewer resources than a run of single-hop tracks and also reduces propagation delay. We found that IF resource utilization decreases linearly as tracks are replaced with long tracks, with routability decreasing at a faster rate. For a given ratio of long to short tracks, resource utilization decreases quadratically and routability decreases exponentially as the length of long tracks is increased, with bases proportional to the ratio of long tracks. These results suggest that long tracks must closely match the needs of a netlist to prevent poor general routability.

Jump tracks are long tracks that are not confined to a single channel, which provide direct connections between distant regions of a fabric. Besides greatly reducing the delay over long routes, jump tracks can also reduce routing congestion over the regular fabric, possibly reducing the need for other routing resources.

Wide channels represent a compromise over increasing track density uniformly across the IF, by increasing track density of only particular channels. Wide channels potentially enable routing for netlists with locally high communication requirements, without the high cost of increasing capacity globally.

Connection box flexibility varies the number of connection boxes and the number of possible connections. Exploration results argue strongly in favor of increasing connection box flexibility for IFs. For example, adding connection boxes on column channels to a fabric results in a 40% increase in routability with only a 10% increase in resource utilization. For a fabric with both, forming connections with both inputs and outputs of CUs, as opposed to connecting to one or the other, results in an additional 30% increase in routability, with only a 15% increase in overhead.

5. EXPERIMENTS

5.1 Experimental Setup

To minimize area overhead of IFs, it was necessary to also assess and minimize the impact on routability. Since IFs are often application-specialized and coarse-grained, existing sets of benchmark netlists used in studies of routing on general-purpose, fine-grained FPGAs [4] are not an ideal method of assessing routability of IFs. Instead, we used a large number of randomly-generated netlists (described below), and assigned each fabric a 'routability score' equal to the percentage of netlists routed successfully. For each fabric, up to 1000 netlists were tested so that the score was reproducible over tests with the same number of different random netlists. This approach provides sufficiently high precision to compare similar fabrics and is not biased by the selection of netlists. Note that this measure is made feasible by the fast PAR achievable on IFs. On FPGAs, the same measure could take months.

To test against netlists representative of common circuits, our random netlist generator created directed acyclic graph structures common to pipelined datapaths of data-parallel applications. For a particular fabric, the random netlist generator selects a random number of technology-mapped cells bounded by the size and CU composition of the fabric. The netlist generator then creates different datapath stages, where each stage consists of a random number of technology-mapped cells, with the requirement that each stage has enough cells to connect at least one output from each cell in the previous stage. Connections between stages were made at random while ensuring that no cell would be left without at least one path to the next stage. The generator can thus produce anything from single- to *n*-stage pipelines, as well as multiple disjoint pipeline structures.

To enable rapid exploration of IFs, we developed a tool capable of generating device-independent VHDL for IFs of the type discussed in Section 3.1. The tool takes a fabric description file as input, which assigns fabric parameters including the size and CU composition of the IF, as well as parameters relating to the specialization techniques discussed in Section 4 including: number of tracks in each channel, length and offset of each track, and placement/connectivity of connection boxes.

The IF PAR tools take as input the same fabric description used by the fabric generator in addition to a technology-mapped netlist.

The output of the IF PAR tools is a bitstream that is loaded by the IF to implement the netlist.

For the device tools, the IF HDL was synthesized using Synplicity Synplify Pro C-2009.03. Xilinx ISE 10.1 was used for placement and routing of the IF HDL and to obtain resource utilization and timing results. Select IFs were implemented and tested on a Xilinx Virtex IV LX100 on a Nallatech H101-PCIXM board.

5.2 Case Studies

This section evaluates PAR speedup and area/routability tradeoffs for IFs specialized for a target application. Rather than specializing for an application domain, which corresponds to the envisioned library IF usage model, we illustrate proof of concept by specializing IFs for individual netlists by manually performing the synthesis usage model from Section 3.3. Specifically, we explore different IF data planes to minimize area overhead while maximizing general routability. The identification of application domains and an appropriate measure of routability for circuits within a domain is left as future work.

To evaluate the potential for specialization of IFs, we manually performed the following methodology, which could easily be automated as part of an IF synthesis tool. First, we used the fabric generator to create a fabric with the minimum number of CUs to implement the target netlist. When choosing between different aspect ratios, we avoided extreme situations such as a fabric with 1 row and 50 columns of CUs. Using 4 tracks per channel as a baseline, we gradually reduced the number of tracks uniformly across the entire fabric until the target netlist failed to route. At this point, we randomly explored replacing tracks with long tracks and jump tracks in addition to reducing the tracks in individual channels. We stopped the exploration when obtaining the smallest fabric that could still route the target netlist.

We evaluated specialized IFs for twelve case studies, which we manually implemented as technology-mapped IF netlists. To determine overhead, we also created VHDL implementations of each example that were implemented directly on the FPGA. For some of the case studies, we evaluated two implementations: one using 16-bit fixed point arithmetic (shown with a FXD suffix) and one using 32-bit float arithmetic (shown with a *FLT* suffix). When not explicitly stated, the circuit used 16-bit fixed point. *Matrix multiply* calculates the inner product of two 8-vectors as the kernel of a matrix multiplication. The netlist requires 15 adders and multipliers. *FIR* is a 12-tap finite impulse response (FIR) filter in transpose form with symmetric coefficients, requiring 23 adders and multipliers. *N-body* represents the computational kernel of an n-body simulation, which calculates the gravitational force exerted on a particle due to a number of other particles in two dimensions. The netlist requires 13 arithmetic operators including adders, multipliers, and a divider. *Accum* is a small netlist that monitors an input stream that and counts the number of times that an input value is less than a specified threshold. The netlist consists of 4 comparators and 3 adders. *Normalize* scales and offsets 8 input values from an input stream every cycle, requiring 8 multipliers and 8 adders. *Bilinear* performs bilinear interpolation on an image, requiring 8 multipliers and 3 adders. *Floyd-Steinberg* performs image dithering using 6 adders and 4 multipliers. *Thresholding* performs automatic image thresholding using 8 comparators and 14 adders. *Sobel* performs the Sobel edge detection on an image using a 3x3 convolution kernel, which requires 11 adders and 2 multipliers. *Gaussian blur* performs Gaussian noise reduction using a 5x5 convolution kernel, which requires 25 multipliers and 24 adders. *Max filter* is an image filter that selects the maximum value in a 3x3 sliding window, whose netlist consists of 8 comparators. *Mean filter* similarly filters an image by averaging the values in a sliding window, for 3 different window sizes (3x3, 5x5, and 7x7). The 3x3 netlist required 8 adders and 1 multiplier, whereas the 7x7 netlist required 48 adders and 1 multiplier.

For the image processing examples based on sliding windows (Sobel, Gaussian, max filter, mean filter), we customized the input stream plane of each IF to use smart buffers capable of efficiently streaming windows from images with a maximum size of 1920x1080. These customized stream planes deliver one window per cycle to the data plane. Smart buffers were also used for the direct FPGA implementations.

Table 1 illustrates IF PAR speedup, area overhead, and clock overhead for each case study. The first major column, *PAR Time*, compares PAR execution times for the specialized IF, the PAR execution time when synthesizing VHDL for each example directly to the FPGA, and the resulting PAR speedup. The results show an average PAR speedup of 275x when using fixed-point operators, and 1112x when using floating-point operators, for an overall average of 554x. The PAR speedup is greater for the floating point circuits because, for fine-grained physical devices, each operator is mapped to hundreds of logic elements, increasing the problem size for FPGA PAR relative to fixed-point operators, which can be mapped directly to DSP units. Note that IF times are identical for floating-point and fixed-point version of each example, which illustrates a key IF advantage: IF PAR times are not affected by the CU complexity. In addition, these IF PAR speedups are pessimistic and represent a lower bound due to the direct FPGA examples ignoring PAR times for other system components. In practice, we have observed that controllers (memory, PCIe, etc.) for a particular board can add 10 to 20 minutes of PAR time to a circuit. *IFs completely avoid these times because the controller components are already included in the fabric*. Therefore, in practice, actual PAR speedup is likely to greater than 1000x.

The second major column in Table 1, *IF Area Overhead*, compares the overhead of fabrics specialized for each of the netlists against individual baseline fabrics not using the specialization techniques (i.e., fabrics of the minimum required size with 4 tracks per channel). *Base* is the overhead of the baseline fabric, as described in Section 4. *Specialized* is the overhead of the specialized fabric. *Savings* is the reduction in overhead achieved by specialization. *RtBase* is the routability score of the baseline fabric *for netlists that utilize every CU in the fabric* (i.e., the maximum sized netlist). *RtSpec* is the retained routability after specialization. On average, the baseline fabrics had an area overhead of 18%, which was reduced to 10% after specialization – an average savings of 45%. Most importantly, the average retained routability of the specialized fabrics was 91%, which suggests that little flexibility was sacrificed to reduce the area overhead. Even in the worst case of Gaussian blur, the retained routability was 58%.

The last major column in Table 1, *IF Clock Overhead*, shows the clock frequency overhead of IFs compared to a direct FPGA implementation for the case study netlists. Note that because the

Table 1: PAR speedup and overhead of case study specialized IFs.

	PAR Time			IF Area Overhead					IF Clock Overhead		
	IF	FPGA	Speedup	Base	Specialized	Savings	RtBase	RtSpec	IF	FPGA	Overhead
Matrix multiply FXD	0.6s	1min 08s	**112x**	16%	**6%**	63%	100%	100%	237 MHz	283 MHz	**16%**
Matrix multiply FLT	0.6s	6min 06s	**602x**	31%	**13%**	58%	100%	100%	249 MHz	224 MHz	**-11%**
FIR FXD	0.6s	0min 33s	**54x**	18%	**12%**	33%	100%	99%	207 MHz	337 MHz	**39%**
FIR FLT	0.6s	4min 36s	**454x**	41%	**29%**	29%	100%	99%	196 MHz	283 MHz	**31%**
N-body FXD	0.5s	0min 57s	**126x**	10%	**5%**	50%	100%	99%	226 MHz	286 MHz	**21%**
N-body FLT	0.5s	3min 42s	**491x**	21%	**10%**	52%	100%	99%	233 MHz	328 MHz	**29%**
AccumFXD	0.1s	0min 26s	**280x**	4%	**2%**	50%	100%	100%	235 MHz	397 MHz	**41%**
Accum FLT	0.1s	0min 30s	**323x**	7%	**5%**	29%	100%	100%	304 MHz	406 MHz	**25%**
Normalize FXD	0.2s	1min 10s	**299x**	12%	**4%**	67%	100%	60%	235 MHz	331 MHz	**29%**
Normalize FLT	0.2s	6min 44s	**1726x**	24%	**14%**	42%	100%	60%	240 MHz	294 MHz	**18%**
Bilinear FXD	0.3s	1min 08s	**230x**	10%	**6%**	40%	100%	97%	221 MHz	162 MHz	**-36%**
Bilinear FLT	0.3s	8min 48s	**1784x**	21%	**14%**	33%	100%	97%	217 MHz	296 MHz	**27%**
Floyd-Steinberg FXD	0.1s	1min 27s	**621x**	7%	**4%**	43%	100%	100%	224 MHz	177 MHz	**-27%**
Floyd-Steinberg FLT	0.1s	5min 37s	**2407x**	14%	**10%**	29%	100%	100%	215 MHz	249 MHz	**14%**
Thresholding	1.4s	0min 33s	**24x**	15%	**10%**	33%	100%	99%	202 MHz	347 MHz	**42%**
Sobel	0.3s	2min 28s	**500x**	12%	**6%**	50%	100%	99%	115 MHz	151 MHz	**24%**
Gaussian Blur	3.3s	3min 19s	**60x**	41%	**24%**	41%	96%	58%	108 MHz	115 MHz	**6%**
Max Filter	0.2s	1min 16s	**444x**	9%	**4%**	56%	100%	98%	117 MHz	151 MHz	**23%**
Mean Filter 3x3	0.2s	2min 30s	**962x**	9%	**3%**	67%	100%	100%	115 MHz	147 MHz	**22%**
Mean Filter 5x5	1.9s	3min 25s	**110x**	21%	**13%**	38%	100%	95%	100 MHz	135 MHz	**26%**
Mean Filter 7x7	8.9s	5min 03s	**34x**	42%	**26%**	38%	95%	59%	101 MHz	129 MHz	**22%**
Average FXD	1.3s	1min 49s	**275x**	16%	**9%**	48%	99%	90%	175 MHz	225 MHz	**18%**
Average FLT	0.3s	5min 09s	**1112x**	23%	**14%**	39%	100%	94%	236 MHz	297 MHz	**19%**
Average	1.0s	2min 56s	**554x**	18%	**10%**	45%	100%	91%	195 MHz	249 MHz	**18%**

specialized IFs use a pipelined interconnect, all netlists for each fabric execute at the specified frequency, which is a function of CU and track propagation delays. To avoid an underestimation of IF overhead, we maximized the performance of the direct FPGA implementations by manually pipelining the direct implementations, using cores configured identically to the fabric's CUs. Average clock overhead was similar for both floating-point and fixed-point examples, with an overall average of 18% that corresponds to 195 MHz. A few examples were actually faster on the IF than when implemented directly on the FPGA. Although this situation should in theory never occur, we have observed some inherent randomness in PAR tools that can occasionally cause similar situations.

Although 18% may already be an acceptable overhead for designers looking for PAR speedup and/or portability, in many situations such overhead will actually be neglible due to other system bottlenecks. For example, components external to the design netlist (e.g., memory/PCIe controllers) often prevent such high frequencies from being obtained. In these situations, the IF clock overhead would effectively have no performance overhead.

Due to space constraints, we omit a detailed analysis of each specialized fabric and instead summarize the results. On average, the specialized fabrics used 2.14 tracks per row channel and 1.86 tracks per column channel. None of the fabrics required more than 3 tracks in any channel, and all but two of the examples (matrix and mean5x5) used less than 3 tracks. When considering that larger examples Gaussian blur and mean7x7 required less tracks, matrix multiply and mean5x5 could likely use less tracks by exploring different aspect ratios. All examples except thresholding and Sobel used a mix of high-flexibility and low-flexibility connection boxes in different parts of the fabric. Three examples used long tracks, and four examples used wide channels to eliminate localized routing bottlenecks.

5.3 General Purpose Analysis

In this section we evaluate the feasibility of IFs as general purpose fabrics, by exploring tradeoffs between overhead and routability for different fabric sizes, and by modifying the data plane to eliminate the pipelined interconnect. Because IFs are not necessarily intended for general purpose usage and will typically be specialized, these results represent a worst case scenario.

5.3.1 Overhead/routability tradeoff analysis

Table 2 compares overhead and routability for different fabric sizes, different track densities, and different connection box flexibilities. All fabrics use Xilinx DSP48s as CUs. The fabric sizes range from 3x3 to 9x9, in addition to a 12x8 fabric that utilizes all 96 of the DSP48 units on the Virtex4 LX200. *OH%* is the overhead of the fabric as defined in previous sections. *Rt%* is the routability in terms of percentage of random netlists that can be successfully routed on the fabric. *RtFull%* is similar, but only considers maximum-sized netlists that use every CU in the fabric (e.g., 96 CUs for the 12x8 fabric). In each cell, the left number represents a low flexibility connection box that only connects inputs or outputs to an adjacent channel. The right number represents a connection box that can connect all CU I/O to an adjacent routing track.

These results again show the effectiveness of flexible connection boxes, resulting in scores of 96% vs. 63%, for fabrics with 2 tracks per channel. The results also suggest that for the pipelined datapath circuits that are the focus of this paper, 4 or 5 tracks per channel is excessive, resulting in a larger overhead without a significant improvement in routability. The most reasonable tradeoffs for general purpose usage are 2 tracks-per-channel with high-flexible connection boxes, or 3 tracks-per-channel with low-flexibility connection boxes. For the 2 tracks-per-channel fabric,

Table 2: Overhead and routability tradeoffs for various sized general-purpose intermediate fabrics with both low-flexibility connection boxes (numbers on left) and full flexibility (numbers on right).

	2 Tracks per Channel			3 Tracks per Channel			4 Tracks per Channel			5 Tracks per Channel		
Size	**OH%**	**Rt%**	**RtFull%**	**OH%**	**Rt%**	**RtFull%**	**OH%**	**Rt%**	**RtFull%**	**OH%**	**Rt%**	**RtFull%**
3x3	2/3	100/100	99/100	4/5	100/100	100/100	5/6	100/100	100/100	6/9	100/100	100/100
4x4	5/6	97/100	84/100	7/10	100/100	99.5/100	9/12	100/100	100/100	12/16	100/100	100/100
5x5	8/10	93/100	83/100	12/15	100/100	99/100	15/18	100/100	100/100	19/25	100/100	100/100
6x6	11/15	91/100	70/99.5	17/22	100/100	96/100	21/27	100/100	100/100	28/36	100/100	100/100
7x7	16/20	87/99	69/97	24/30	98/100	94/100	30/37	100/100	100/100	38/50	100/100	100/100
8x8	21/25	87/99	51/92	31/39	99/100	91/100	39/48	100/100	100/100	50/65	100/100	100/100
9x9	29/32	81/98	37/88	40/49	97/100	88/100	49/59	99.9/100	98/100	64/82	100/100	100/100
12x8	31/37	79/97	12/89	46/57	98/99.9	90/99	57/68	99.6/100	98/100	75/95	100/100	99.8/100
Average	**15/19**	**89/99**	**63/96**	**23/28**	**99/100**	**94/100**	**28/34**	**100/100**	**99/100**	**37/47**	**100/100**	**100/100**

even at a size of 12x8 (96 DSP units), the overhead is only 37% of the device. Although there are applications that may require a larger fabric, there are numerous applications that require less than 96 DSP operations. In fact, the Virtex IV LX family does not have a device with more than 96 DSPs. By mapping DSP operations onto LUTs after using all 96 DSP units, we were able to implement a 15x15 (225 16-bit CUs) fabric on the Virtex 4 LX 200, which is large enough to enable many embedded usage scenarios.

5.3.2 IF routing delays

In this section we consider the effects of eliminating the pipelined interconnect in order to support arbitrary circuits required for general purpose usage.

Static timing analysis presents one interesting challenge for the IF tool flow. For a non-pipelined IF, timing is not known until the IF is placed and routed onto a physical device. Therefore, timing data must be back annotated after FPGA PAR for IF PAR to make accurate decisions. For example, FPGA PAR tools could place two adjacent CUs onto resources located at opposite ends of the FPGA. To reduce this problem, FPGA placement of different IF components can be constrained to use specific areas of the FPGA when possible (e.g., FPGAs often lay out multipliers in columns). We leave such optimization for future work and instead present pessimistic results obtained by letting the FPGA PAR tools place each IF CU.

Table 3 illustrates propagation delays of IF routing resources for two baseline fabrics, showing modest increases with fabric size. We report propagation delays instead of clock frequency because for the general purpose fabrics, clock frequency is dependent on the number of routing resources required by a particular netlist. For a given netlist on the general purpose fabric, the clock frequency is determined by the sum of the delays of IF tracks. Unlike FPGAs, which have somewhat uniform delays for similar routing resources, IFs may have significantly different delays due to variation in the FPGA PAR process. Thus, one IF track may be able to run at 195 MHz, while all others could potentially run faster. Exploiting non-symmetric delays for tracks will require specialized placement algorithms, which we leave as future work.

Table 3: Track delays for general-purpose baseline IFs.

	Max Common Clock	Mean Track Delay	Min Track Delay	Max Track Delay
Baseline 5x5	4.9 ns	1.17 ns	0.62 ns	2.89 ns
Baseline 9x9	4.9 ns	1.42 ns	0.70 ns	3.02 ns

6. LIMITATIONS AND FUTURE WORK

Even with the specialization techniques presented in the paper, IFs will often not scale to large circuits that can be implemented in state-of-the-art FPGAs. However, there are numerous usage scenarios that do not require circuits of these sizes, such as ASIPs, accelerators for embedded kernels, and dynamic binary synthesis. In addition, future work focusing on directly mapping IF routing resources onto physical routing resources may significantly decrease overhead and enable new usage models. Furthermore, for many applications, a design may include numerous smaller IFs, which reduces area overhead compared to one large IF.

For many of the specialization techniques discussed, routability could be improved with novel PAR algorithms. Enhancing routability may reduce the required number of IF routing resources, enabling further reduction of overhead. New PAR algorithms will also be required to support more specialized IFs, such as those not using island-style architectures.

Realization of the usage models discussed in Section 3.3 will require future work to determine methodologies for populating IF libraries, algorithms for selecting appropriate IFs from a library, and algorithms for IF synthesis.

7. CONCLUSIONS

In this paper, we introduced intermediate fabrics, which are virtual reconfigurable architectures that represent an intermediate translation layer between a user netlist and a physical device. Intermediate fabrics enable portability of circuits across different physical devices, which can reduce design complexity while also allowing third party design tools to target different devices. In addition, intermediate fabrics reduce complexity of physical devices, which greatly reduces placement and routing times compared to COTS FPGAs, resulting in an average PAR speedup of 554x for 12 case studies. The main limitation of intermediate fabrics is area overhead incurred by virtual routing resources. However, we showed that for reasonably large fabrics with 96 DSP units, the overhead requires approximately 1/3 of a Virtex 4 LX200 while routing 97% of randomly generated pipelined-datapath netlists. In addition, we presented specialization techniques to reduce this overhead for specific domains, which on average reduced overhead by 45% while retaining a routability of 91%. Future work on intermediate fabrics implemented directly on physical FPGA routing resources may eliminate much of this overhead, potentially enabling more usage scenarios.

8. ACKNOWLEDGMENTS

This work was supported in part by the I/UCRC Program of the National Science Foundation under Grant No. EEC-0642422. The authors gratefully acknowledge vendor equipment and/or tools provided by Altera, Nallatech, and Xilinx.

9. REFERENCES

[1] 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.

[2] A. C. S. Beck and L. Carro, "A VLIW low power java processor for embedded applications," in *SBCCI '04: Proceedings of the 17th Symposium on Integrated Circuits and System Design*. New York, NY, USA: ACM, 2004, pp. 157–162.

[3] J. Becker, T. Pionteck, C. Habermann, and M. Glesner, "Design and implementation of a coarse-grained dynamically reconfigurable hardware architecture," in *VLSI '01: Proceedings of IEEE Computer Society Workshop on VLSI*, May 2001, pp. 41–46.

[4] V. Betz. The FPGA place and route challenge. http://www.eecg.toronto.edu/~vaughn/challenge/challenge.html.

[5] V. Betz and J. Rose, "VPR: A new packing, placement and routing tool for FPGA research," in *FPL '97: Proceedings of the 7th International Workshop on Field-Programmable Logic and Applications*. London, UK: Springer-Verlag, 1997, pp. 213–222.

[6] T. J. Callahan, P. Chong, A. DeHon, and J. Wawrzynek, "Fast module mapping and placement for datapaths in 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. 123–132.

[7] K. Compton and S. Hauck, "Totem: Custom reconfigurable array generation," in *FCCM'01: Proceedings of the the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines* 2001, pp. 111–119.

[8] S. Craven and P. Athanas, "Examining the viability of FPGA supercomputing," *EURASIP J. Embedded Syst.*, vol. 2007, no. 1, pp. 13–20, 2007.

[9] A. DeHon, "The density advantage of configurable computing," *Computer*, vol. 33, no. 4, pp. 41–49, 2000.

[10] Y. Dong, Y. Dou, and J. Zhou, "Optimized generation of memory structure in compiling window operations onto reconfigurable hardware," in *ARC*, 2007, pp. 110–121.

[11] C. Ebeling, D. C. Cronquist, and P. Franklin, "Rapid - reconfigurable pipelined datapath," in *FPL '96: Proceedings of the 6th International Workshop on Field-Programmable Logic,Smart Applications, New Paradigms and Compilers*. London, UK: Springer-Verlag, 1996, pp. 126–135.

[12] K. Eguro and S. Hauck, "Armada: timing-driven pipeline-aware routing for FPGAs," in *FPGA '06: Proceedings of the 2006 ACM/SIGDA 14th international symposium on Field programmable gate arrays*, 2006, pp. 169–178.

[13] R. Fung, V. Betz, and W. Chow, "Simultaneous short-path and long-path timing optimization for FPGAs," in *ICCAD '04: Proceedings of the 2004 IEEE/ACM International Conference on Computer-aided design*, 2004, pp. 838–845.

[14] Z. Guo, B. Buyukkurt, andW. Najjar, "Input data reuse in compiling window operations onto reconfigurable hardware," in *LCTES '04: Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, 2004, pp. 249–256.

[15] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau, "Spark: a high-level synthesis framework for applying parallelizing compiler transformations," in *VLSI'03: Proceedings of the 16th International Conference on VLSI Design,* Jan. 2003, pp. 461–466.

[16] M. Hammerquist and R. Lysecky, "Design space exploration for application specific FPGAs in system-on-a-chip designs," in *SOC '08: Proceedings of the IEEE International SOC Conference*, Sept. 2008, pp. 279–282.

[17] Impulse Accelerated Technologies, Inc. Impulse CoDeveleoper. 2010. http://www.impulseaccelerated.com/products.htm.

[18] A. K. Jones, R. Hoare, D. Kusic, J. Fazekas, and J. Foster, "An fpga-based vliw processor with custom hardware execution," in *FPGA '05: Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, 2005, pp. 107–117.

[19] N. Kapre, N. Mehta, M. deLorimier, R. Rubin, H. Barnor, M. J. Wilson, M. Wrighton, and A. DeHon, "Packet-switched vs. time-multiplexed FPGA overlay networks," in *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, 2006.

[20] A. Koch, "Structured design implementation: a strategy for implementing regular datapaths on FPGAs," in *FPGA '96: Proceedings of the 1996 ACM fourth international symposium on Field-programmable gate arrays*, 1996, pp. 151–157.

[21] 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*, 2004, pp. 954–959.

[22] 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*, 2005, pp. 57–62.

[23] A. Marshall, T. Stansfield, I. Kostarnov, J. Vuillemin, and B. Hutchings, "A reconfigurable arithmetic array for multimedia applications," in *FPGA '99: Proceedings of the 1999 ACM/SIGDA Seventh International Symposium on Field Programmable Gate Arrays*, 1999, pp. 135–143.

[24] L. McMurchie and C. Ebeling, "Pathfinder: a negotiation-based performance-driven router for FPGAs," in *FPGA '95: Proceedings of the 1995 ACM Third International Symposium on Field Programmable Gate Arrays*, 1995, pp. 111–117.

[25] S. Merchant, et al., "Strategic challenges for application development productivity in reconfigurable computing," in *NAECON '08: Proceedings of the IEEE National Aerospace and Electronics Conference*, July 2008.

[26] Mitrionics, Inc. The Mitrion Virtual Processor. 2010. http://www.mitrionics.com/?page=mitrion-virtual-processor.

[27] 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*, 2001, pp. 29–36.

[28] B. E. Nelson, M. J. Wirthlin, B. L. Hutchings, P. M. Athanas, and S. Bohner, "Design productivity for configurable computing," in *ERSA '08: Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms*, 2008, pp. 57–66.

[29] NSF Center for High-Performance Reconfigurable Computing. FPGA tool-flow studies workshop hosted by CHREC. June 2008. http://www.chrec.org/ftsw/.

[30] L. Sekanina, *Evolvable Systems: From Biology to Hardware*. Springer Berlin / Heidelberg, 2003, ch. Virtual Reconfigurable Circuits for Real-World Applications of Evolvable Hardware, pp. 116–137.

[31] A. Sharma, S. Hauck, and C. Ebeling, "Architecture-adaptive routability-driven placement for FPGAs," *International Conference on Field Programmable Logic and Applications*, pp. 427–432, 2005.

[32] S. Shukla, N. W. Bergmann, and J. Becker, "Quku: A two-level reconfigurable architecture," in *ISVLSI '06: Proceedings of the IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures*, 2006, p. 109.

[33] G. Stitt and F. Vahid, "Energy advantages of microprocessor platforms with on-chip configurable logic," *IEEE Design & Test*, vol. 19, no. 6, pp. 36–43, 2002.

[34] W. Tsu, K. Macy, A. Joshi, R. Huang, N. Walker, T. Tung, O. Rowhani, V. George, J. Wawrzynek, and A. DeHon, "HSRA: high-speed, hierarchical synchronous reconfigurable array," in *FPGA '99: Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*, 1999, pp. 125–134.

[35] F. Vahid, G. Stitt, and R. Lysecky, "Warp processing: Dynamic translation of binaries to FPGA circuits," *Computer*, vol. 41, no. 7, pp. 40–46, July 2008.

[36] J. Wang, Q. Chen, and C. Lee, "Design and implementation of a virtual reconfigurable architecture for different applications of intrinsic evolvable hardware," *Computers & Digital Techniques, IET*, vol. 2, no. 5, pp. 386–400, September 2008.