Fine-Grained Interconnect Synthesis

ALEX RODIONOV, University of Toronto
DAVID BIANCOLIN, University of California at Berkeley
JONATHAN ROSE, University of Toronto

One of the key challenges for the FPGA industry going forward is to make the task of designing hardware easier. A significant portion of that design task is the creation of the interconnect pathways between functional structures. We present a synthesis tool that automates this process and focuses on the interconnect needs in the fine-grained (sub-IP-block) design space. Here there are several issues that prior research and tools do not address well: the need to have fixed, deterministic latency between communicating units (to enable high-performance local communication without the area overheads of latency insensitivity), and the ability to avoid generating unnecessary arbitration hardware when the application design can avoid it. Using a design example, our tool generates interconnect that requires 69% fewer lines of specification code than a handwritten Verilog implementation, which is a 32% overall reduction for the entire application. The resulting system, while requiring 6% more total functional and interconnect area, achieves the same performance. We also show a quantitative and qualitative advantages against an existing commercial interconnect synthesis tool, over which we achieve a 25% performance advantage and 15%/57% logic/memory area savings.

CCS Concepts: • **Networks → Network on chip**; *Topology analysis and generation*; • **Hardware → High-level and register-transfer level synthesis**; *Hardware accelerators;*

Additional Key Words and Phrases: FPGA, interconnect, automated synthesis

ACM Reference Format:
Alex Rodionov, David Biancolin, and Jonathan Rose. 2016. Fine-grained interconnect synthesis. ACM Trans. Reconfigurable Technol. Syst. 9, 4, Article 31 (August 2016), 22 pages.
DOI: http://dx.doi.org/10.1145/2892641

1. INTRODUCTION

An important and time-consuming aspect of hardware design is creating the interconnect that allows computation, storage, and control logic to communicate. This interconnect is often nontrivial, as any need for arbitration, routing, or pipelining precludes the use of wires alone. Furthermore, an application's communication requirements may evolve throughout its development life cycle, requiring effort and time to change the interconnect.

Interconnect synthesis tools have already been developed to automate the generation and parameterization of different kinds of interconnect, increasing designer productivity by avoiding the tedious and error-prone process of manually (re)writing an RTL description [Altera Corporation 2015; Xilinx Corporation 2015; Lattice Semiconductor

Authors' addresses: A. Rodionov and J. Rose, Department of Electrical and Computer Engineering, University of Toronto, 10 King's College Road, Toronto, Ontario, Canada M5S 3G4; emails: arod@eecg.toronto.edu, Jonathan.Rose@ece.utoronto.ca; D. Biancolin, Department of Electrical Engineering and Computer Science, University of California at Berkeley, 592 Soda Hall, Berkeley, CA 94720; email: biancolin@eecs.berkeley.com.

DOI: http://dx.doi.org/10.1145/2892641

2015; Papamichael and Hoe 2012; Huan and DeHon 2012]. Additionally, some of these tools also perform system integration and instantiate and parameterize the designer's functional modules, as well as connect them together using the automatically generated interconnect, further reducing designer effort.

In the FPGA sphere, these existing tools are primarily focused on system-level design, connecting processors with peripherals, hardware accelerators, and other large chunks of IP. A key element of this design paradigm is latency insensitivity [Carloni et al. 2001], in which some form of Valid and Ready handshaking signals are used by the interface between the designer's modules and the interconnect to allow variable latency and backpressure. This decouples the interconnect's interface from its implementation, enabling IP reuse and drop-in replacement of interconnect (e.g., switching from a crossbar to an on-chip network). Owing to their processor-centric roots, other common features of these interface protocols include memory-mapped addressing and support for read and write transactions.

However, with the increasing complexity of FPGA applications, IP blocks are starting to contain their own internal hierarchy and interconnect, which existing tools are not well equipped to generate. This fine-grained design environment is qualitatively characterized by the relatively small size of the blocks being connected, as well as an increased sensitivity to communication latency. Together, these factors make the area and performance overheads of coarse-grained interconnect prohibitive. For example, a pipelined network-on-chip router with virtual channels and a complex routing algorithm would be excessive for connecting together a handful of modules that are smaller than the router itself.

Additionally, coarse-grained interface paradigms such as latency insensitivity, memory mapping, and read/write transactions can incur secondary performance and area penalties if conforming to such interfaces forces the designer to insert extraneous logic. For example, supporting variable latency and backpressure requires pipelined datapaths to contain FIFOs or staging registers (two registers and a mux) instead of chains of ordinary registers, to avoid data loss on deassertion of a Ready signal.

In this article, we present a new interconnect synthesis and system integration tool, GENeric Interconnect Engine (GENIE). Our long-term goal is to automate and optimize interconnect for all levels of design granularity, but in this article we will focus on its ability to generate fine-grained interconnect. We will show that the generated interconnect is comparable in area and performance to handcrafted RTL and requires less effort on behalf of the designer to specify.

GENIE's interconnect protocol lies between existing streaming and memory-mapped protocols in its level of abstraction. It defines signal roles for data, flow control, backpressure, and multicast-capable addressing, with most roles being optional. This allows for simple and minimal interfacing on the part of the designer. The tool's generated interconnect network is made of cascading split and merge primitives, which have been shown [Huan and DeHon 2012] to exhibit high performance and low area usage in FPGA applications.

To address the need for deterministic latency, GENIE allows the designer to query the latency of generated interconnect and pass it as a Verilog parameter to instantiated compute modules. Combined with GENIE's ability to pipeline interconnect, this aids in design space exploration and the hunt for timing closure, without the need for the designer to manually reparameterize their noninterconnect datapaths. Additionally, GENIE generates smaller and faster versions of its interconnect by completely removing arbitration logic when the designer can guarantee the absence of competition on shared, many-to-one connections. Other features of GENIE, which are helpful and not limited to fine-grained use, include the support for configurable network topologies and optimized automatic clock domain crossing.

The structure of this article is as follows. Section 2 provides relevant background on existing interconnect synthesis tools and paradigms. Section 3 describes the GENIE tool, its features, and interconnect microarchitecture. Section 4 describes the fine-grained design example that we will use to evaluate the capabilities of the tool—a compute unit in a parallel LU matrix decomposition [Zhang et al. 2012] engine.

In Section 5, we generate interconnect for this design example using GENIE, Altera Qsys [Altera Corporation 2015], and hand-optimized Verilog, and compare their area usage and achieved clock frequencies. We also attempt to quantify the productivity gains GENIE offers by comparing the amount of source code required to generate each of the three versions. We conclude our findings in Section 6.

A previous version of this article appears in Rodionov et al. [2015]. This version contains more detail on our automatic clock domain crossing algorithm in Section 3.6, a description and illustration of the tool's ability to experiment with alternative network topologies in Section 4.4, and individual measurements of the area and performance benefits of specific GENIE features in Section 5.3.

2. BACKGROUND

In this section, we provide an overview of existing interconnect synthesis tools and methodologies. When studying existing tools, it is important to consider two aspects when considering applicability for fine-grained synthesis: the designer-facing protocol(s) afforded by the tool and the architecture of the generated interconnect.

Altera and Xilinx include system integration tools with their FPGA design suites. Xilinx's Vivado IP Integrator [Xilinx Corporation 2015] and Altera's Qsys [Altera Corporation 2015] both provide two classes of interconnect protocols: memory mapped and streaming.

Memory-mapped protocols, such as AMBA AXI [ARM Ltd. 2015] and Altera's Avalon-MM, are intended for connecting processors to peripherals and custom accelerators—what we consider to be a coarse-grained design space. Communications must be expressed as byte- or word-addressable reads and writes, between masters (initiators) and slaves (responders). Conforming to this high level of abstraction grants a designer a high degree of interconnect automation, with automatic insertion of data width converters, clock domain crossers, and the routing and arbitration logic necessary for decoding addresses and sharing a slave between multiple masters, respectively. The latter two functions of routing and sharing/arbitration are implemented with a shallow fixed-topology network such as a crossbar or a shared bus.

Meanwhile, streaming protocols such as AXI-Stream [ARM Ltd. 2015] and Avalon-ST are on the low end of the automation spectrum, with the intent mainly to provide a consistent IP interface rather than enable automated synthesis. Streaming protocols allow specification of only point-to-point connections, which consist of nothing more than data and flow control (Valid and Ready) signals, in typical use. The interconnect is implemented as point-to-point wiring, and the designer must explicitly insert any IP cores for routing and arbitration to communicate with multiple endpoints. This allows great control over implementation at the cost of increased design effort.

Recent academic work in interconnect synthesis for FPGAs has focused on automatic generation of on-chip networks. CONNECT [Papamichael and Hoe 2012] is an interconnect architecture specifically designed for the FPGA fabric, operating faster and with less area than direct ports of ASIC-targeted architectures. An online, Web-based generator allows users to create custom networks with arbitrary topologies and architectural features such as number of virtual channels. However, CONNECT does not perform system integration, and the designer is responsible for instantiating the interconnect and connecting it to their functional modules.

The interconnect architecture presented in this article is based on Split-Merge [Huan and DeHon 2012], another existing FPGA network-on-chip design. Instead of traditional monolithic routers, simple split and merge primitives are used to implement one-to-many and many-to-one communications, respectively. These can be chained together to form arbitrary topologies, giving a designer more implementation control than the crossbar-based memory-mapped interconnect provided by FPGA vendor tools.

Algorithms have been developed to synthesize application-specific network topologies from a high-level specification consisting of connectivity and bandwidth/latency/energy requirements [Pinto et al. 2003; Ogras and Marculescu 2005; Todorov et al. 2014]. In particular, one approach [Cong et al. 2011] removes complexity from the generated network if it is known a priori that certain communication traces will never occur simultaneously. We perform a similar optimization during our network generation flow.

3. GENIE

We now describe GENIE, our new system integration and interconnect synthesis tool. Its input is a logical specification of a system's desired communication links and its output is a synthesizable Verilog implementation of the system that instantiates and parameterizes the designer's functional modules and connects them with an automatically generated interconnect fabric. We begin with the detail of GENIE's designer-facing interface protocol, and then move on to the microarchitecture of the generated interconnect, and finally describe other features that ease designer burden in fine-granularity hardware contexts.

3.1. Input Specification

To use GENIE, the designer describes the functional modules to be instantiated and the logical communication links between them. This specification follows the component-based design paradigm used by most other system integration tools [Altera Corporation 2015; Xilinx Corporation 2015; Lattice Semiconductor 2015], which is essentially a higher-level representation of structural hardware design. The designer defines one or more *systems*, each containing instances of *components* that represent Verilog modules.

Each component has one or more *interfaces*, which have a direction of dataflow and a type, and contain one or more Verilog input/output signals. Each signal within an interface is assigned a communication-related role, such as transmitting data, providing an address, or synchronization. Interfaces allow components to communicate with other components and with hardware outside the system. An interface's type specifies its communication protocol, and GENIE defines the following types of interfaces:

Clock, *Reset*: Delivers clock (or reset) signals to the component.

Routed Streaming: Serves as an endpoint for GENIE's routed streaming communications protocol and can include a mix of data, handshaking, addressing, and packetization signals, further explained in Section 3.2. An associated clock interface determines the clock domain.

Conduit: A catch-all for signals that wish to bypass GENIE's interconnect synthesis flow, such as those connecting to off-chip memory controllers. Connecting together two conduits connects together their constituent signals with simple wires.

After instantiating components within a system, the designer defines *links* between their interfaces. These are logical connections representing desired communication paths and are made between interfaces of the same type and opposite data flow direction. A system also contains one or more *exports*, which enable communication into and out of the system. Exports are connected to interfaces via links, and the export

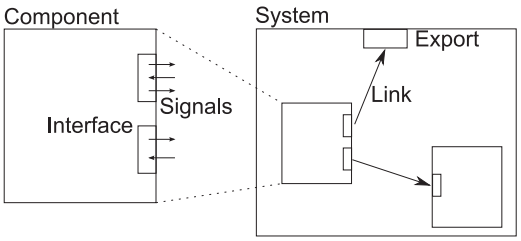

Fig. 1. GENIE specification.

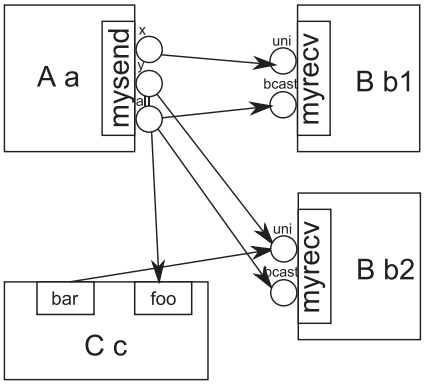

Fig. 2. Example system.

Table I. Linkpoint Definitions for Example System

Component	Interface	LP Name	LP ID
A	mysend	x	2'b00
		y	2'b01
		all	2'b10
B	myrecv	uni	1'b0
		bcast	1'b1
C	foo	—	—
	bar	—	—

automatically takes on the same type and opposite polarity of its connected counterpart. Figure 1 provides an overview of the objects in GENIE's system specification.

GENIE also offers an addressing scheme that sits on top of the interface abstraction, which allows an interface to choose a subset of outgoing links to transmit data, or allows a receiving interface to be notified of which link is currently sending it data. This is achieved through abstractions called *linkpoints*. Each routed streaming interface, which is a physical collection of signals, may have one or more named linkpoints defined inside it, which are virtual connection points associated with the physical interface. A linkpoint is simply a name and an associated binary encoding (a linkpoint ID) chosen by the designer, used to refer to the linkpoint by the component's logic. At the system level, links are normally made between two interfaces to indicate logical connectivity. When an interface has linkpoints, a link must terminate at one of the linkpoints instead. During circuit operation, a component drives (or receives) a linkpoint ID to differentiate among multiple remote destinations.

The example system illustrated in Figure 2 shows a GENIE input specification and the use of linkpoints. In this example, components A, B, and C are Verilog modules created separately by the designer. In the system specification, modules A and C are instantiated once (named as instances `a` and `c`), and B is instantiated twice as instances `b1` and `b2`. Components A, B, and C each have routed streaming interfaces, named in the rectangular labels within A, B, and C, and are listed in the second column of Table I. Associated clock and reset interfaces are omitted for clarity. Components A and B use interfaces that also include linkpoints (shown as external circles in Figure 2), whereas C does not need to differentiate among destinations and thus does not use the linkpoint addressing scheme. The third column of Table I gives the linkpoint names, with their associated linkpoint IDs (in Verilog notation) given in the fourth column.

The result is that instance a can send data to either b1, b2 (using the x→uni or y→uni connections) or broadcast to b1 and b2 and c simultaneously (using its all outgoing linkpoint). Module instances b1 and b2 can differentiate between received unicast and broadcast traffic, and take take appropriate action if they wish.

Note that linkpoint IDs are defined during component definition, effectively creating a local rather than global address space, thus removing the need for the designer to write additional address encoding/decoding logic. The ability to broadcast/multicast can have important application in coarse-granularity designs, but we have also found a natural use case for it in our fine-granularity design example described in Section 4. There it is used to selectively fill several block RAMs simultaneously.

All of these specifications are written by the designer programmatically, in the form of an executable script written in Lua [PUC-Rio 2015]. The script makes API calls to our underlying interconnect synthesis engine that create interface, component, system, link, export, and linkpoint definitions. In a future version of the tool, interface definitions and signal roles will be extracted directly from Verilog signal definitions of each component's source code, removing the current need by GENIE to replicate this information.

3.2. Communication Protocol

An important decision in creating a useful interconnect automation tool is the choice of the level of abstraction for the designer-facing interface protocol, which is dependent on the intended use of the tool. If there is insufficient abstraction and automation, then the designer must implement some interconnect functionality explicitly—for example, streaming protocols are extremely *lightweight*, but any arbitration or routing logic must be inserted manually, as the protocol has no concept of an address.

On the other hand, a protocol can be too *heavyweight*—for example, tools that synthesize memory-mapped interconnect allow masters to address different slaves and automatically insert the logic to route traffic accordingly. This includes enabling sharing and arbitration from competing masters. The price of this automation is that it requires the designer to express all communications as byte-addressable reads and writes, even in situations where it is unnatural to do so.

In the fine-granularity design space, which is the focus of this article, we wish to elevate the level of automation above that of bare RTL and streaming protocols but avoid the overhead of memory-mapped interconnect. We landed on using a streaming protocol but augmenting it with the optional ability to reach different destinations using the linkpoint addressing scheme described earlier. The tool automatically inserts lightweight logic to perform the requisite routing and arbitration. We call this a *routed streaming* protocol.

The routed streaming protocol defines several roles for the signals that constitute a routed streaming interface:

—*Data*: The data to transmit, of arbitrary designer-specified width. There can be several independent data signals within the interface—for example, to carry red, green, and blue color data separately if one is transmitting pixel data. This saves the designer from having to manually pack and unpack data fields in their component logic. Multiple data signals must be differentiated with a designer-provided tag.
—*Valid*: Indicates whether all other signals carry valid values during the current clock cycle.
—*Ready*: Backward-traveling backpressure signal that indicates if the interconnect, or designer logic, is able to accept data during a clock cycle.

—*EOP*: End of packet. Used for transmitting a large block of data over multiple cycles and is asserted on the last cycle. The interconnect uses this signal for arbitration purposes.
—*LPID*: The linkpoint ID, if any linkpoints are defined for this interface.

The direction of each signal matches the direction of the interface, except for Ready signals, which travel in the opposite direction. Most signals are optional, and GENIE avoids the generation of unnecessary logic when signals are left unused. The minimal possible interface consists of either only a Data signal or only a Valid signal. The latter case is useful for implementing dataless messages such as *go* or *done* commands issued by control logic. When a Valid, Ready, or EOP signal is omitted, the interface behaves as if there is a constant high value driving that signal.

3.3. Interconnect Microarchitecture

In addition to a lightweight interface protocol, it is important that the generated interconnect have low area overhead and introduce minimal latency, especially in fine-granularity systems. GENIE's interconnect is based on two switching primitives called *split* and *merge* [Huan and DeHon 2012]. They perform all routing and arbitration functionality and can be cascaded to form arbitrary topologies.

The ability to control topology is an important feature when designing networks. GENIE allows the designer to specify topology on a per-system level as a parameter when declaring a system in the input script. There exist several built-in topologies, such as sparse crossbar, ring, and shared bus. These are topology-generating functions that create the correct number of split and merge nodes depending on the system specification, and the designer can write his or her own generator function in Lua to implement custom topologies. We provide an example in Section 4.4.

GENIE's interconnect employs static, table-based routing, which is generated based on the logical links defined by the designer in the input specification. Internally, GENIE assigns a global flow ID to each end-to-end link that was specified by the designer. Table-based converters are inserted in front of routed streaming interfaces of components to convert between designer-defined linkpoint IDs and global flow IDs.

The conversion is expressed as a logic function, and we found that it optimizes to wires and one or two FPGA logic elements during back-end synthesis.

Routing is performed by split nodes (as illustrated in Figure 3), which have a single input and multiple outputs. The data payload is broadcast to all destinations and includes routed streaming signals such as Data and EOP that the split node does not need to extract/examine. A split node contains a table, parameterized by GENIE, which looks up a one-hot vector of Valid outputs for each flow ID. These Valid signals pass through an Eager Fork [Carmona et al. 2009] stage. The Eager Fork is a substructure that is part of the split node and is responsible for throttling Valid signals and managing state when only a subset of currently targeted outputs are ready to receive the broadcasted payload. It also breaks combinational loops when split nodes are cascaded with merge nodes [Carmona et al. 2009].

Merge nodes (shown in Figure 4) allow multiple input streams to compete for one common output. A round-robin arbiter selects which input gets forwarded to the output. If multicycle packets are being sent, and the EOP signal is being used, then the merge node will wait until the entire packet is sent, and the EOP received, before switching inputs. This eliminates the need for the designer's logic to track multiple overlapping transmissions and the resulting complexity involved. If the inputs can be guaranteed to never simultaneously access the output, the round-robin arbiter can be removed (Section 3.5).

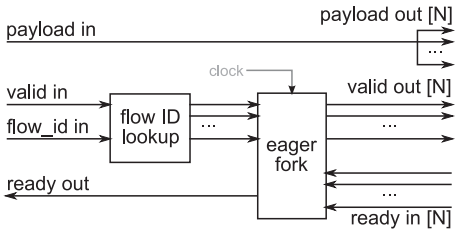

Fig. 3. Split node architecture.

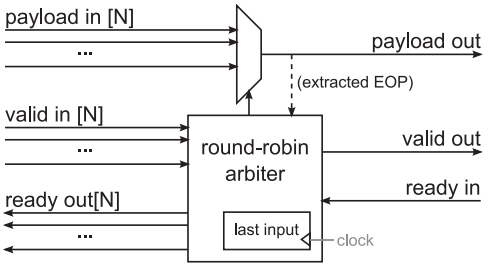

Fig. 4. Merge node architecture.

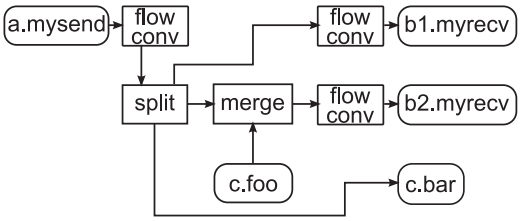

Fig. 5. Example system with interconnect.

Figure 5 shows a GENIE implementation of the example system specification from Figure 2. Interfaces are shown with rounded boxes, and the regular boxes are GENIE interconnect (flow ID converters, split nodes, and merge nodes). Clock and reset connections are omitted for clarity.

3.4. Latency Introspection

In the context of high-performance functional element design (which we have labeled the "fine-grained" context), the latency of communication paths must often be short and of deterministic length. For example, it is common to have highly pipelined datapaths performing some computation. Although the interconnect between individual pipeline stages is currently outside the scope of our synthesis flow, the communication between the pipeline's exterior with control logic or with a pipelined block RAM does fit into the fine-granularity realm that we wish to target.

In the latter case, where data signals temporarily leave the pipeline to access a block RAM for reading, the read data must be reunited with associated data that stay within the pipeline—and those signals must be delayed by the correct amount. That amount depends on the RAM read latency plus the delay of the interconnect. If a traditional, latency-insensitive interface is used, the designer's pipeline must be able to tolerate backpressure, which introduces additional complexity and possibly even FIFOs, thus incurring area, performance, and design time overheads.

Instead, we would like the interconnect, just like the block RAM being accessed, to have a deterministic and fixed latency, but one that can still be trivially modified later to ease timing closure. GENIE solves this problem with its *latency introspection* feature, which allows the actual interconnect latency to be queried and reported back to components' Verilog code as parameter values. Queries are made during system definition in the Lua script, and values are propagated during component instantiation.

3.5. Mutually Exclusive Sharing

In general many-to-one communication, there must exist some method for the interconnect to allow two or more inputs to share a common destination. When there is

competition, arbitration logic must choose the winner and stall the other inputs with backpressure. This necessitates support for backpressure on all competing links and their upstream sources. In GENIE, sharing and arbitration are accomplished with the merge node.

However, if the application is deliberately designed such that no two inputs will ever simultaneously access their shared destination, arbitration is no longer necessary. Such mutually exclusive access patterns can arise, for example, when each competing source has explicitly scheduled access to the destination. When a designer creates a GENIE system specification, he or she can also specify a constraint indicating that all links terminating at a shared destination will never compete during application runtime.

This generates a simplified merge node with the round-robin logic in Figure 4 removed. In its place, the select input of the multiplexer is driven directly (in a one-hot fashion) by the incoming Valid signals, which are also ORed together to generate the outgoing Valid signal. The Ready signal is broadcast to all of the inputs.

3.6. Automatic Clock Domain Crossing

Multiple clock domains are often used to decouple computation and communication circuitry if they have unbalanced demands. GENIE transparently supports multiple clock domains and inserts crossing logic automatically in the form of dual-clock FIFOs.

When a design contains multiple clock domains, there is an interesting optimization problem that arises when crossing between any two domains: where in the generated interconnect network should the transition occur? GENIE intelligently chooses the point at which the minimum total number of signals undergo the crossing, because each signal incurs a nontrivial cost. For example, consider that building the crossing before the input to a split node is cheaper than inserting multiple crossings after each output of the split node. When the network contains a complex topology of split and merge nodes, the optimal choice may not be obvious.

The crossing-point selection algorithm presented in Algorithm 1 represents the system as a directed graph $G = (V, E)$. Vertices represent routed streaming interfaces belonging to both designer-specified functional modules and those of internally generated interconnect primitives. $C(v)$ represents the clock domain of each vertex, and initially only some vertices will have this assignment, from a set of domains K. The objective of the algorithm is to find the best clock domain assignment for each of the remaining unlabeled vertices.

Directed edges between the vertices represent connectivity between the corresponding interfaces. Each edge e has a weight $W(e)$ that represents the cost of placing a clock domain crossing there. This cost is a function of the number of data bits on that link plus the nominal fixed overheads of maintaining a dual-clock FIFO such as read/write pointer registers. The clock domain assignment problem is formulated as a multiway-cut [Dahlhaus et al. 1992] problem: to partition the graph into connected components (one for each clock domain) while minimizing the total weight of the boundary edges between them. This problem is NP-hard for more than two clock domains, and Algorithm 1 is based on a greedy approximation.

The algorithm requires that each clock domain be represented with a single *terminal vertex*. We create each terminal in the set T by merging together all vertices that share the corresponding clock domain. The vertex merging procedure is explicitly laid out in Algorithm 2. We then assign one clock domain at a time by repeatedly finding a minimum cut between one of the terminal vertices t_i and a merged vertex representing all other terminal vertices s_0. This is accomplished with a standard min-cut (dual of max-flow) algorithm that returns the total weight of the minimal cut $cost_i$ and a residual graph R_i. A greedy decision is made to choose the terminal vertex that yields the least-cost two-way cut. The vertices in that terminal's partition are assigned its

ALGORITHM 1: Area-Optimal Clock Domain Assignment

inputs: connectivity G = (V,E), edge weights W, *partial* clock assignments C, clock domains K
output: clock assignments C for *all* $v \in V$

```
T := {}
// Collapse all vertices that share a clock domain, add them to T
foreach k ∈ K do
    U_k := (u_0, u_1, u_2, ... ∈ V | C(u_i) = k) // all ports driven by clock k
    MergeVertices(G, U_k)
    T := T ∪ {u_0} // one terminal vertex per clock domain
end
// Assign clock domains one at a time, removing them from a copy of G as we go
H := G
while |T| > 1 do
    // Try the terminal for each remaining unassigned clock domain
    foreach t_i ∈ T do
        // Merge the terminals for all the other domains into a single vertex s_0
        H' := H
        U := (s_0, s_1, ... ∈ T | s_j ≠ t_i)
        MergeVertices(H', U)
        // Find the min-cut between source t_i and sink s_0
        // Memoize the residual graph R_i and total cut weight cost_i
        R_i, cost_i := MinSTcut(H', W, t_i, s_0)
    end
    // Choose the clock domain terminal that yielded the smallest-weight cut
    cost_best := smallest cost_i
    // Assign all reachable vertices the corresponding clock domain
    foreach v ∈ R_best reachable from t_best do
        C(v) := C(t_best)
        remove v from H
    end
    T := T\{t_best}
end
// Only one clock domain remains, do a trivial assignment
foreach v ∈ H do
    C(v) := C(the one member of T)
end
```

ALGORITHM 2: MergeVertices(G, U)

inputs: G = (V,E), list of vertices to merge U=$(u_0, u_1, u_2, \dots)$
output: updated G with vertices from U merged into u_o

```
E := E\{(x, y) | x, y ∈ U} // remove edges between members of U
// Transfer outgoing edges to u_0
foreach (x, y) ∈ {E | x ∈ U_k ∧ y ∉ U_k} do
    E := E\{(x, y)}
    E := E ∪ {(u_0, y)}
end
// Repeat for incoming edges
// ...
V := V\{u_1, u_2, u_3, ...}
```

corresponding clock domain and are removed from the graph. With one fewer unassigned clock domain, the process repeats until all vertices are assigned a domain.

Finally, after the algorithm terminates and all interfaces have an assigned clock domain, GENIE inserts clock-crossing FIFOs at clock domain boundaries. This feature

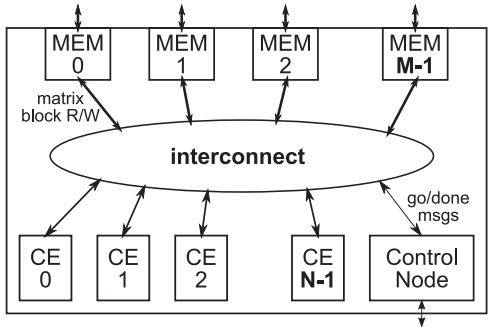

Fig. 6. LU decomposition engine.

of GENIE contributes to rapid design exploration—in addition to evaluating different topologies, a designer can also experiment with assigning functional modules to different clock domains to try to optimize application performance, all without modifying the application's RTL source code.

4. DESIGN EXAMPLE

In this section, we present a hardware design example that, in Section 5, will be used to evaluate and compare GENIE's fine-grained interconnect synthesis versus manual design and a commercial interconnect synthesis tool. With this example, we also hope to better illustrate the nature of fine-grained interconnect and the challenges related to automatically synthesizing it.

Our application is a parallel LU decomposition engine [Zhang et al. 2012]. LU decomposition is an important linear algebraic operation and is often used as the first step in efficiently solving systems of linear equations or calculating matrix inverses. It decomposes a square matrix A into lower-triangular and upper-triangular matrices L and U such that $L \times U = A$. The application stores the matrix in off-chip memory so that very large matrices can be decomposed. It is arranged in blocked fashion (64×64) to support blocked computation and partitioned across M memory controllers. An array of N compute elements (CEs), coordinated by a central control node, work in parallel to process the matrix and write back a transformed version in-place. A diagram of the full system is given in Figure 6.

The CEs, memory nodes, and control node are large (using between 1,000 and 10,000 logic elements) and must tolerate variable communication latency, in part due to the nature of external memory. It is at this level that a traditional coarse-grained interconnect synthesis is typically employed. The long-term goals of the GENIE project are to both generate at this level and the fine-grained level that we have described so far, *and* to optimize across all of those levels. However, in this article, we focus on the fine-grained system design within a *single* CE.

4.1. CE Design

Here we describe the structure, functionality, and internal communication requirements of the CE, which is normally instantiated many times within the larger LU decomposition application but is examined in isolation as our design example.

The CE processes a specific column of blocks from the matrix by reading the blocks from external memory and writing back transformed data in their place. A simplified block diagram of the CE is shown in Figure 7 and contains four major components:

—A control unit to orchestrate the fetching, processing, and writing back of blocks within the assigned column.

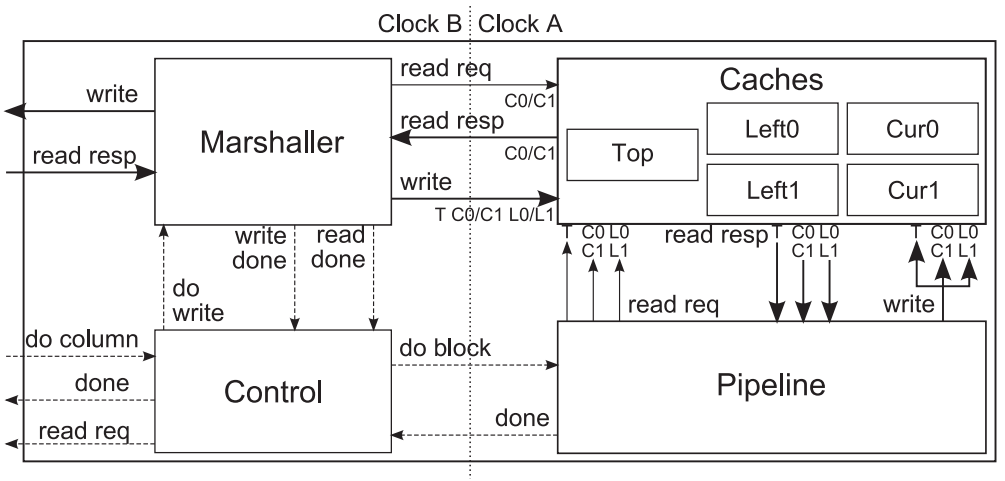

Fig. 7. CE architecture.

—Caches, implemented as FPGA block RAMs, to store the matrix blocks being operated on, locally within the CE.
—A computation pipeline, which reads from and writes to the caches to produce the processed results.
—A data marshaller to transfer matrix blocks to and from the caches and external memory outside the CE.

There are five independent, dual-ported cache blocks in total: Top, Left0, Left1, Current0, and Current1. They are named after the types of blocks they store during processing and relate to the spatial relationships between the cached blocks within the larger matrix. The Left and Current blocks are also double buffered for increased performance, with the numerical suffix indicating to which buffer it belongs. While the caches of one buffer are being processed by the pipeline, the other buffer is being filled from, or written back to, main memory by the marshaller. The Top block is rarely written to and does not need a second buffer.

The CE has two clock domains to decouple the performance requirements of the two tasks of processing matrix blocks and transferring them to and from memory. Processing a block takes much longer, as each element in the matrix must be accessed more than once, on average. The pipeline and caches operate using “Clock A,” and the rest of the design uses “Clock B,” including the coarse-grained interconnect linking the CE with the greater LU decomposition system.

There are two kinds of communications present within the CE shown in Figure 7: low-throughput control messages (shown as dashed arrows), and high-throughput matrix block read requests, read replies, and writes (shown as solid arrows). The former, although being point to point and not performance demanding, can still benefit from automated interconnect synthesis rather than being implemented by hand, either because of the need to cross clock domains (control to pipeline) or the potential need to pipeline the links to close timing later in the design cycle.

In contrast, the high-throughput communications links, whose logical connectivity is depicted in Figure 8, require high-performance and nontrivial interconnect. They send data words (or requests for data words) every cycle, and originate or terminate at the read or write port of one of the five caches. For all but two links (pipeline to top cache reads), some mix of one-to-many or many-to-one communications is needed,

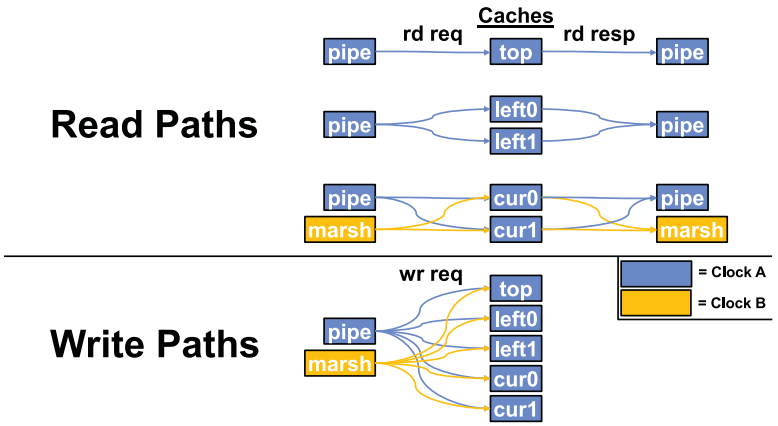

Fig. 8. Logical connectivity between marshaller, pipeline, and caches.

requiring distribution or arbitration hardware within the interconnect. The pipeline and marshaller perform writes of identical data to some subset of the caches, which changes at runtime, and thus requires either multicast capability or multiple write ports.

The double buffering of the caches is explicitly managed by the application and guarantees that the marshaller and pipeline *never* compete for the same cache's read or write port. This application-specific behavior presents opportunities to optimize the design of the interconnect to reduce area and increase performance.

Read requests are 12 bits wide and specify an address within a cache. Read replies are 256 bits wide and carry multiple words of data to feed the pipeline's SIMD datapath. Writes contain both an address and data and are 268 bits wide. It is important to mention the relatively large width of these connections, as it makes the interconnect's area usage that much more sensitive to its architecture.

4.2. Three CE Implementations

To illustrate the power, flexibility, and quality of results of our new approach, we created three different implementations of the system and its interconnect: one generated by our tool GENIE, a manually written and optimized reference design, and one generated by Altera's Qsys [Altera Corporation 2015] system integration tool. This allows us to compare GENIE against the best possible hardware (at the expense of design time) and against an existing automated synthesis tool (at the expense of performance).

Each variant is a different realization of the CE system shown in Figure 7. In the Qsys variant, two different communication protocols are used: Avalon-MM (memory mapped), and Avalon-ST (streaming). Connections to the caches map naturally to random-access reads and writes, so we implement those using Avalon-MM, using an extra address bit to select between buffers of double-buffered caches. The remaining connections, which are point to point and have no memory-like semantics, are implemented using Avalon-ST.

In the GENIE variant, all connections in Figure 7 are implemented as links defined between interfaces using the routed streaming protocol. Interfaces with multiple fanout, such as those to and from the caches, have linkpoints defined for each possible combination of destinations. The address is part of the data payload rather than being an official part of the designer-facing interface as with Avalon-MM, so the purpose of linkpoints is simply to direct traffic to the correct cache and buffer.

The manual variant uses no synthesis tool and contains application-specific interconnect, designed by hand, to implement functionality such as connection sharing or clock crossing. Pipeline registers were also manually added to improve timing on specific paths.

4.3. Tool-Related Issues

In this section, we highlight some important differences between the interconnect implementations of each variant to give some context to the results in Section 5. The goal of automation is to improve designer productivity while generating hardware with acceptable area and performance. To that end, we also hope to provide a qualitative picture of the design effort required to create each variant.

To avoid a detailed and exhaustive comparison, we focus on how each variant handles the following aspects of the CE design, as they required the greatest interconnect complexity:

—Clock domain crossing
—Marshaller to cache read path
—Pipeline to cache read paths
—Cache write paths

4.3.1. Clock Domain Crossing. Both the marshaller-to-cache connections and the control-to-pipeline connections cross clock domain boundaries, which is handled differently among the three variants.

In the manual variant, there exist two clock-crossing FIFOs for the whole design: one for connections traveling from Clock A to Clock B and one in the other direction. Each FIFO handles multiple links that travel in the same direction. This is the most efficient implementation and is specifically tailored for the application.

Qsys performs automatic clock crossing for Avalon-MM connections, inserting dual-clock FIFOs when a master and slave are on different clock domains. However, it inserts FIFOs *after* routing traffic to multiple destinations, causing each destination path to have its own FIFO, including nine FIFOs that must accommodate the cache read/write data width (256+ bits). Finally, no automatic clock crossing is performed on the Avalon-ST connections for the low-bandwidth control messages, requiring manual instantiation of clock crossing adapters from the Qsys component library.

The GENIE implementation has one clock-crossing FIFO for each connection (for a total of five) rather than the two used in the manual variant. All marshaller-to-cache write paths share a single FIFO, which GENIE inserts before a split node that broadcasts to up to five caches. This was determined using the algorithm described in Section 3.6. The total number of FIFO memory bits is thus identical to the manual variant, but there is extra logic overhead since each FIFO requires its own read/write pointer and metastability protection registers. The upside is that all routed streaming connections receive automated clock crossing, with no designer intervention needed.

4.3.2. Marshaller-to-Cache Reads. Cache reads from the marshaller need to be able to stall if the system outside the CE is unable to accept the outgoing data during any given clock cycle.

The manual variant's caches have an explicit "stall" signal as an input, which is generated by the marshaller directly rather than being locally derived from any kind of backpressure conditions.

In the GENIE variant, the caches have flow control and backpressure (Valid and Ready) signals on both read request and read response ports, and are able to stall the block RAM's internal pipeline if the read data is not accepted by the marshaller.

The Avalon-MM protocol has backpressure for read requests in the form of the `waitrequest` signal role, allowing slaves (the cache read ports) to stall the master (the marshaller). There exists no signal that allows the marshaller to stall read data returning from the caches. Our solution was to add a FIFO to the marshaller to buffer this data until it can be sent outside the CE, and reserving space in this FIFO before sending any read requests to the caches. Note that this missing functionality in Qsys requires extra effort for the designer to mitigate while also costing area. This is not a general limitation of memory-mapped protocols, as, for example, AMBA AXI [ARM Ltd. 2015] has backpressure support for request and reply paths, but is cumbersome to use since many signals are mandatory.

4.3.3. Pipeline-to-Cache Reads. Read and write access to (some) of the cache blocks is shared between the pipeline and marshaller. However, due to double buffering of the caches, and careful orchestration by the control logic, the design of the CE guarantees no competition between the pipeline and marshaller for the same buffer. This is ideal, because in theory it allows the pipeline to operate as if it has sole point-to-point access to the caches with the benefit of deterministic latency, simplifying the design.

This is the case in the handmade variant. Sharing of the cache ports is done with muxes controlled directly by the control logic, which guarantees that the marshaller and pipeline never access the same cache buffers at the same time. At the read data output of the caches, a simple mux chooses the correct buffer's read response stream to send back to the pipeline. The round-trip latency of the read path is fixed and known by the pipeline, allowing it to use simple register chains to delay other signals, instead of a more expensive latency-insensitive construct such as a FIFO.

GENIE's implementation of the read path is similar to the handmade variant. The specification for the read request/response paths contains a hint that the marshaller and pipeline will never simultaneously compete for the same cache ports, resulting in simplified merge nodes that are equivalent to the muxes of the handmade variant. The effective difference is that these muxes are controlled locally (by the incoming Valid signals) rather than centrally by the control logic. Using latency introspection (described in Section 3.4), the pipeline is able to know the exact fixed latency of the GENIE-generated read path interconnect and can avoid using FIFOs, just like the handmade version.

The Qsys interconnect inserts arbiters that allow both the marshaller and pipeline to access the read ports of the caches. These arbiters are designed for the general worst case in which competition is always a possibility, increasing their cost relative to the GENIE and handmade implementations. The pipeline also had to be modified to use a latency-insensitive FIFO to delay signals alongside the read path, as the Qsys interconnect always asserted backpressure during the first cycle of a burst transfer, preventing a completely smooth flow of data. This modification to the pipeline resulted in more complicated hardware than the latency-sensitive register chain in the other two implementations.

4.4. Network Topology

As mentioned in Section 3.3, the split and merge nodes that comprise GENIE's physical interconnect can be arranged to form many different topologies. For many applications, it is enough to choose one of GENIE's built-in topology functions, which require no additional input from the designer. However, it is also possible to create custom network topologies to exploit communication patterns in the application.

The default topology used by GENIE is its built-in *sparse crossbar* topology. It is programmatically generated according to two rules:

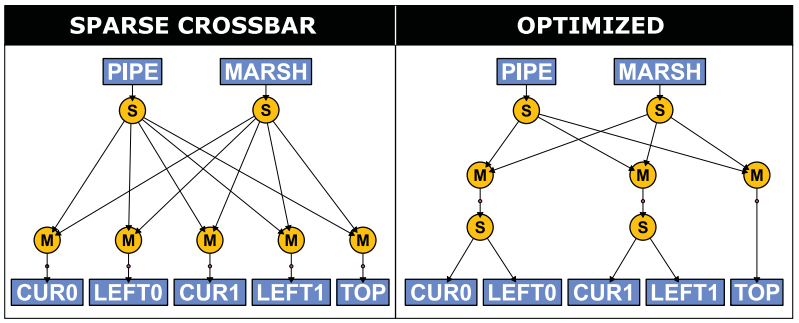

Fig. 9. Sparse crossbar (default) and optimized (application-specific) topology implementations for write requests. The networks are built from split (S) and merge (M) nodes, with smaller circles representing postmerge register stages.

(1) Every source with multiple sinks generates a split node.
(2) Every sink with multiple sources generates a merge node and a register stage.

This scheme is also known as slave-side arbitration, and it has the property that competition for network bandwidth occurs only at the sink, as there exists a dedicated physical path for each source-to-sink logical path. This is the scheme used by Qsys interconnect, with different arbitration/distribution primitives in place of GENIE's split and merge nodes, and is the only available option.

However, the pipeline and marshaller are guaranteed to never compete for cache access in our application, and this can be taken advantage of to create the area-optimized topology for cache write request links shown in Figure 9. Compared to a crossbar topology, it reduces the number of merge nodes by two without creating any contention points in the network. These merge nodes are of the simplified mutually exclusive type in both cases, but since write requests are 268 bits wide, the removal of two of them still represents nontrivial logic and register savings, which will be quantified in Section 5.3. The additional two split nodes incur minimal overhead, as their cost is independent of payload width. All other links in the GENIE variant of the system are realized with the default crossbar topology, as it already yields the minimal possible number of split and merge nodes.

This fine tuning of topology design would normally only be possible with handcrafted Verilog, as is the case with our manual CE variant. GENIE provides a much simpler alternative through writing a custom topology function while reusing the same logical link specification, thereby allowing fine-grained design optimization and topology exploration without giving up the convenience of automation.

5. RESULTS

In this section, we quantitatively compare the three CE variants presented in Section 4 to judge the efficacy (and ease of use) of GENIE in generating fine-grained interconnect.

Automation should increase productivity and make life easier for the designer. The implementation issues discussed in Section 4.3 give a qualitative view of the designer effort required. Here, we measure the amount of source code (and tool specification code) line counts as a first-order quantitative approximation of the difficulty of creating each CE variant.

At the same time, automation should strive to produce a high-quality interconnect implementation. We obtain the area and F_{max} of each variant after being synthesized, placed, and routed on a modern FPGA.

Table II. Code Line Counts: Designer Effort

Variant	I/C + TOP	FUNC	TOTAL
Manual	1,029	1,323	2,352
GENIE	314	1,289	1,603
Qsys	440	1,411	1,851
GENIE vs. Manual	–69%	–3%	–32%
GENIE vs. Qsys	–29%	–9%	–13%

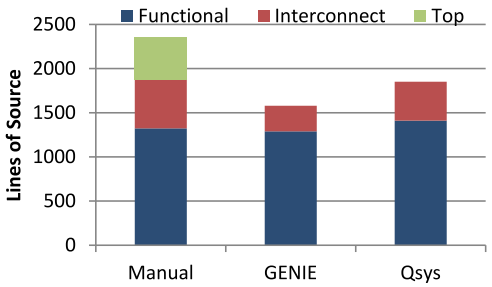

Fig. 10. Code line counts.

After comparing the three implementations, we also show the effect of the different optimizations within four different GENIE-produced variants to shed light on the value of the optimizations.

5.1. Source Code Line Count

First, we measure the number of lines of source code (including scripting input lines for the tools) required to create both the functional modules and interconnect for each variant. For the interconnect part, we are interested in the size of the specification directly written by the designer. For the Qsys and GENIE variants, this would be the size of the scripts (written in TCL and Lua, respectively) that are given as input to the tools to describe the system's communicating components and logical connectivity. The manual variant's interconnect is written in Verilog, as are the functional modules in all three variants.

Line counts were obtained using the CLOC [2015] tool, which ignores comments and blank lines. Table II and Figure 10 give the results, with the table showing both absolute and relative line counts.

Note that the manual variant's 1,029 lines of interconnect source code include 481 lines solely dedicated to the top-level Verilog module that instantiates all of the other modules; this is referred to as TOP in the table and figure. This glue code does not specify any true functionality yet comprises a large portion of the source code base. Figure 10 gives it its own category to provide a better comparison of "real" interconnect specification size, which is 548 lines for the manual variant. Nevertheless, using either system integration tool spares the designer from having to manually write the top-level instantiation code, so we include it together with interconnect in the I/C + Top category in Table II.

The design of the functional modules is also affected by the choice of interconnect synthesis tool, to be compatible with protocols or mitigate lack of features, as described in Section 4.3. The GENIE variant requires 3% less functional code than the manual variant, with minor architectural changes. It also required 9% less functional code than the Qsys variant, which required the more significant changes described in Section 4.3.

Table III. Clock Frequencies

Variant	Clock A	Clock B
Manual (MHz)	406	461
GENIE (MHz)	400	496
Qsys (MHz)	320	394
GENIE vs. Manual	–1%	+8%
GENIE vs. Qsys	+25%	+26%

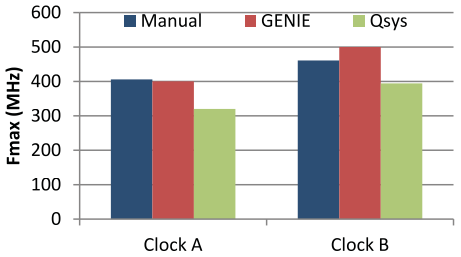

Fig. 11. Clock frequencies.

In the end, if a designer were to create the CE with GENIE in mind from the very beginning, he or she would need to write 32% less source code than with no automation at all, with an even greater reduction of 69% if we focus on just the interconnect. The respective savings over Qsys are 12% (total) and 29% (interconnect only). It is a crude, but quantifiable, measurement of savings in design effort.

5.2. Area and Clock Frequency

Synthesis of each variant was performed using Altera Quartus II version 14.0, targeting a large Stratix V 5SGX-MBBR-1H43-C2 device, with the expectation of low congestion and device utilization. All external signals terminate at virtual inputs/outputs rather than real device pins. Both clock domains in the design were overconstrained to 1GHz, and results were geometrically averaged over six random seeds.

Table III and Figure 11 show the achieved frequency for both clock domains for each variant and a relative comparison of GENIE against the other two variants.

GENIE's interconnect achieves a Clock A frequency only 3% slower than the manual variant's. In the Clock B domain, GENIE achieves a 8% frequency advantage, but at the cost of extra registers. Since our Stratix V device's frequency is limited to 450MHz anyway, a design choice was made in the manual variant to use fewer register stages—a detailed level of control that we hope to include in a later revision of GENIE.

Like the handmade interconnect, GENIE is able to take advantage of the application-level optimization that allows zero competition for caches and thus generates very similar connection-sharing hardware. The simplified merge nodes are one reason why, against Qsys, GENIE performs 25% better on average.

Table IV and Figure 12 provide the area usage of the three variants, in terms of Stratix V adaptive logic modules (ALMs) (representing logic, registers, and distributed memory) and M20K memory blocks. All variants also use eight DSP (hard multiplier) blocks in addition to what is shown.

The GENIE-generated system is only 6% larger than the manually created one, and it occupies 15% fewer ALMs than the Qsys-generated system.

The Qsys interconnect contains an overabundance of clock-crossing FIFOs (as discussed in Section 4.3), as well as additional FIFOs used to buffer cache read data. The increased number of FIFOs, and the fact that the GENIE and manual variants

Table IV. Area Usage

Variant	ALM	M20K
Manual	6,739	40
GENIE	7,149	40
Qsys	8,383	92
GENIE vs. Manual	+6%	+0%
GENIE vs. Qsys	–15%	–57%

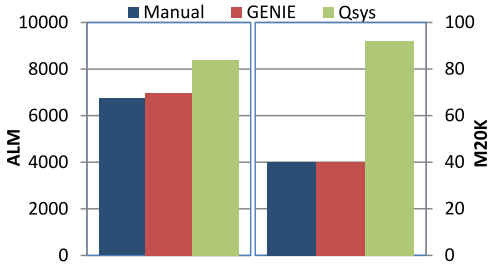

Fig. 12. Area usage.

Table V. Clock Frequency, Area, and Source Code Effects of GENIE Optimizations

Configuration	Clock A (MHz)	Clock B (MHz)	Area (ALMs)	I/C Code (Lines)
NONE	323	504	7,565	280
TOPO	360	480	7,222	290
MERGE	394	497	7,485	304
BOTH	400	496	7,149	314
TOPO vs. NONE	+11%	–4.9%	–4.5%	+24
MERGE vs. NONE	+22%	–1.4%	–1.1%	+10
BOTH vs. NONE	+24%	–1.5%	–5.5%	+34

use distributed memory instead of M20Ks for their FIFOs, explains the high observed M20K usage.

Using M20Ks instead of distributed memory for FIFOs increases ALM usage, making the area gap between Qsys and manual/GENIE narrower than it otherwise might have been. Distributed memory uses ALMs itself and cannot pack as many downstream pipeline registers as M20Ks can. These registers then go on to use additional ALMs.

5.3. Effects of Application-Specific Interconnect Optimizations

In this section, we measure the performance and area benefits of two of GENIE's optimizations that exploit the application-guaranteed mutual exclusivity of double-buffered communication paths found in our design example:

—Simplified, arbiter-less merge nodes (Section 3.5)
—Custom network topology for cache write requests (Section 4.4)

Table V presents the clock frequency, area, and source code line counts of four GENIE-generated CEs. The configurations named TOPO and MERGE have, respectively, only the custom network topology enabled, and then only the simplified merge nodes enabled. The NONE configuration has neither, and the BOTH configuration has both. The latter is also the configuration used for all previous GENIE results in this section.

The results show that simplified merge nodes mainly provide a performance gain, increasing the achieved compute clock by 22% over the base case while offering a small

area benefit of 1.1% fewer ALMs. This optimization requires only 10 extra lines of Lua interconnect specification code to mark the related links as being mutually exclusive.

In contrast, the custom topology's main contribution is reducing ALM usage by 4.5% over the base case while simultaneously providing a smaller but still significant 11% increase in compute clock frequency. The reported network clock is negatively affected but is still well above Quartus's restricted F_{max} cap of 450MHz for Stratix V. This optimization costs the designer 24 extra lines of code to write the custom topology function.

Together, both optimizations provide a 24% increase in compute clock frequency and a 5.5% reduction in ALM usage and require 34 more lines of Lua code than the base case. Without them, the achieved performance of the GENIE-generated CE would be similar to that of the Qsys variant reported in Table III. Exploitation of communication patterns is necessary for GENIE to automatically create fine-grained interconnect that approaches the area and performance of handcrafted Verilog.

Software Release

GENIE is open source software and is available for download at http://www.eecg.utoronto.ca/~jayar/software/GENIE/, complete with documentation and design examples, including Lua input specification scripts. The most recent version has been massively re-engineered, resulting in a higher-quality code base that will enable the research described next.

6. CONCLUSIONS AND FUTURE WORK

We have presented a new interconnect synthesis and system integration tool and showed its applicability in a fine-granularity design space that has been neglected by existing tools. We showed how to express interconnect requirements used this tool to synthesize the interconnect for a realistic fine-grained design example. This was compared with a hand-implemented version, as well as one made with a commercial interconnect synthesis tool.

Qualitatively, we found that the new tool, GENIE, reduced design effort by automating aspects of design such as clock crossing, as well as the generation of the switching fabric that allows the functional modules to communicate. All of this was done without significant changes to the functional modules to support GENIE's protocol. This is in contrast with Qsys, which required changes to the functional modules to use, mitigating a lack of features in the signal protocol.

Quantitatively, using GENIE resulted in a 32% reduction in total source code line count compared to the handmade implementation and a significant 69% reduction if one only considers the code required to specify the interconnect. The cost for this productivity gain was a modest 3% decrease in achieved clock frequency (in one of the two clock domains) and a 6% increase in area. This demonstrates that the automation and ease of use provided by the tool, our primary goal, does not detract from the interconnect's performance in a frugal fine-granularity design context.

Against Qsys, GENIE achieved clock frequency gains of 25% and 26% in the CE's two clock domains, and a 15% reduction in logic and register usage. The RAM block count reduction was more significant, at 57%. These gains demonstrate the efficacy of GENIE's automatic clock crossing insertion algorithm and lightweight Split-Merge interconnect microarchitecture in a fine-granularity design.

We examined two of GENIE's optimizations that depended heavily on exploiting application-specific communication patterns: mutually exclusive sharing (which creates simplified merge nodes) and customizable topologies (which can reduce area and/or increase clock frequency with respect to generic topologies). Our single design example was able to take advantage of both of them, together providing a 24% clock frequency

improvement and a 5.5% reduction in area. These optimizations took advantage of double buffering, which is a common enough design technique and communication pattern that we foresee these optimizations being useful in other applications as well.

Although this article focused on its fine-granularity use, we envision GENIE as a contender for interconnect synthesis at all levels of design, including the creation of the packet-switched networks and memory-mapped interconnect from the efficient split and merge primitives already in use. By having a single tool responsible for generating interconnect at all levels, it will be possible to explore new techniques, such as optimizing interconnect across hierarchy boundaries.

In the future, we plan to expand GENIE's capabilities in the coarse-grained interconnect domain and move it toward an optimization platform that accepts constraints on the performance of the interconnect and makes decisions for the designer that reduce cost while meeting these constraints. We expect that these optimizations will be simultaneously at both fine- and coarse-grained levels.

REFERENCES

Altera Corporation. 2015. QSys—Altera's System Integration Tool. Retrieved June 30, 2016, from http://www.altera.com/products/software/quartus-ii/subscription-edition/qsys/qts-qsys.html.

ARM Ltd. 2015. AMBA Specifications. Retrieved June 30, 2016, from http://www.arm.com/products/system-ip/amba/amba-open-specifications.php.

L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli. 2001. Theory of latency-insensitive design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 20, 9, 1059–1076. DOI:http://dx.doi.org/10.1109/43.945302

J. Carmona, J. Cortadella, M. Kishinevsky, and A. Taubin. 2009. Elastic circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 28, 10, 1437–1455. DOI:http://dx.doi.org/10.1109/TCAD.2009.2030436

CLOC. 2015. CLOC: Count Lines of Code. Retrieved June 30, 2016, from http://cloc.sourceforge.net/.

Jason Cong, Yuhui Huang, and Bo Yuan. 2011. A tree-based topology synthesis for on-chip network. In *Proceedings of the 2011 IEEE/ACM International Conference on Computer-Aided Design (ICCAD'11)*. IEEE, Los Alamitos, CA, 651–658. DOI:http://dx.doi.org/10.1109/ICCAD.2011.6105399

E. Dahlhaus, D. S. Johnson, C. H. Papadimitriou, P. D. Seymour, and M. Yannakakis. 1992. The complexity of multiway cuts (extended abstract). In *Proceedings of the 24th Annual ACM Symposium on Theory of Computing (STOC'92)*. ACM, New York, NY, 241–251. DOI:http://dx.doi.org/10.1145/129712.129736

Yutian Huan and A. DeHon. 2012. FPGA optimized packet-switched NoC using split and merge primitives. In *Proceedings of the 2012 International Conference on Field-Programmable Technology (FPT'12)*. 47–52. DOI:http://dx.doi.org/10.1109/FPT.2012.6412110

Lattice Semiconductor. 2015. LatticeMico System Development Tools. Retrieved June 30, 2016, from http://bit.ly/1fsLLj6.

U. Y. Ogras and R. Marculescu. 2005. Energy- and performance-driven NoC communication architecture synthesis using a decomposition approach. In *Proceedings of the Design, Automation, and Test in Europe Conference*, Vol. 1. 352–357. DOI:http://dx.doi.org/10.1109/DATE.2005.137

Michael K. Papamichael and James C. Hoe. 2012. CONNECT: Re-examining conventional wisdom for designing NoCs in the context of FPGAs. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA'12)*. ACM, New York, NY, 37–46. DOI:http://dx.doi.org/10.1145/2145694.2145703

Alessandro Pinto, Luca P. Carloni, and Alberto L. Sangiovanni-Vincentelli. 2003. Efficient synthesis of networks on chip. In *Proceedings of the 21st International Conference on Computer Design (ICCD'03)*. 146–150.

PUC-Rio. 2015. The Programming Language Lua. Retrieved June 30, 2016, from http://www.lua.org/.

Alex Rodionov, David Biancolin, and Jonathan Rose. 2015. Fine-grained interconnect synthesis. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'15)*. ACM, New York, NY, 46–55. DOI:http://dx.doi.org/10.1145/2684746.2689061

V. Todorov, D. Mueller-Gritschneder, H. Reinig, and U. Schlichtmann. 2014. Deterministic synthesis of hybrid application-specific network-on-chip topologies. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 33, 10, 1503–1516. DOI:http://dx.doi.org/10.1109/TCAD.2014.2331556.

Xilinx Corporation. 2015. Accelerating Integration. Retrieved June 30, 2016, from http://www.xilinx.com/products/design-tools/vivado/integration/.

Wei Zhang, Vaughn Betz, and Jonathan Rose. 2012. Portable and scalable FPGA-based acceleration of a direct linear system solver. *ACM Transactions on Reconfigurable Technology and Systems* 5, 1, Article No. 6. DOI:http://dx.doi.org/10.1145/2133352.2133358

Received August 2015; revised December 2015; accepted February 2016