# Fine-Grained Interconnect Synthesis

Alex Rodionov, David Biancolin, and Jonathan Rose
The Edward S. Rogers Sr. Department of Electrical & Computer Engineering
University of Toronto
{arod, Jonathan.Rose}@ece.utoronto.ca, biancolin@eecs.berkeley.edu

## ABSTRACT

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 un-necessary arbitration hardware when the application design can avoid it. Using a design example, our tool generates interconnect that requires 72% fewer lines of specification code than a hand-written Verilog implementation, which is a 33% overall reduction for the entire application. The resulting system, while requiring 4% 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 17%/57% logic/memory area savings.

## Categories and Subject Descriptors

B.5.2 [**Register-Transfer-Level Implementation**]: Design Aids—*automatic synthesis*

## Keywords

FPGA, interconnect, automated synthesis

## 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 non-trivial, since any need for arbitration, routing, or pipelining precludes the use of wires alone.

Furthermore, an application's communication requirements may evolve throughout its development lifecycle, 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[1, 15, 9, 12, 8]. Additionally, some of these tools also perform *system integration* and instantiate and parameterize the designer's functional modules as well as connecting 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*[3], 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 re-use and drop-in replacement of interconnect (switching from a crossbar to an on-chip network, for example). 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, in order to avoid data loss upon deassertion of a Ready signal.

In this paper, we present a new interconnect synthesis and system integration tool called GENIE (GENeric Interconnect Engine). Our long-term goal is to automate and optimize interconnect for *all* levels of design granularity, but in this paper 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 hand-crafted 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 [8] 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 re-parameterize their non-interconnect 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 paper 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[16] engine.

In Section 5, we generate interconnect for this design example using GENIE, Altera Qsys[1], 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. Finally, we conclude our findings in Section 6.

## 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[15] and Altera's Qsys[1] both provide two classes of interconnect protocols: memory-mapped and streaming.

Memory-mapped protocols, such as AMBA AXI[2] 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[2] 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[12] 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 paper is based on Split/Merge[8], 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[10, 11, 14]. In particular, the approach[6] used by Cong et al. 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

In this section we 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 which instantiates and parameterizes the designer's functional modules as well as connecting 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 which ease designer burden in fine-granularity hardware contexts.

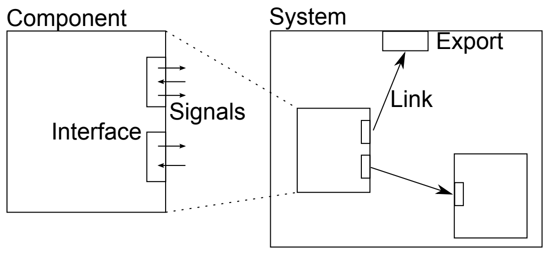

**Figure 1: GENIE Specification**

## 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 [1, 15, 9] which is essentially a higher-level representation of structural hardware design. The designer defines one or more *Systems*, each containing instances of *Components* which represent Verilog modules.

Each Component has one or more *Interfaces*, which have a direction of data flow 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 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, that allows an Interface to choose a subset of outgoing Links to transmit data, or allows a receiving Interface 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 of 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 amongst multiple remote destinations.

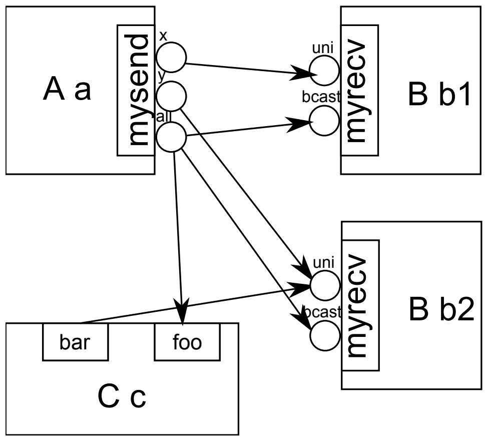

**Figure 2: Example System**

| 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 | - | - |

**Table 1: Linkpoint definitions**

The example System illustrated in Figure 2 shows a GENIE input specification, and the usage of Linkpoints. In this example, components A, B, 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 1. Associated Clock and Reset Interfaces are omitted for clarity. Components A and B use Interfaces that include also Linkpoints (shown as external circles in Figure 2) while C does not need to differentiate amongst destinations and thus does not use the Linkpoint addressing scheme. The third column of Table 1 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 imporant application in coarse-granularity designs, but we have also found a natural use case for it in our fine-granularity design example described below in Section 4. There it is used to selectively fill several block RAMs simultaneously.

All of these specifications are written by the designer programatically, in the form of an executable script written in Lua[13]. 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, since the protocol has no concept of an address.

On the other hand, a protocol can be too *heavyweight* - for example, tools which 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 paper, 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 above. 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**: Backwards-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 data-less messages such as *go* or *done* commands issued by control logic. When a Valid, Ready, or EOP signal is ommitted, the Interface behaves as if there is a constant high value driving that signal.

## 3.3 Interconnect Micro-Architecture

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 [8]. 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 crossbar, ring, and shared bus. These are topology-generating functions, which create the correct number of split and merge nodes depending on the System specification, and the designer can write their own generator function in Lua to implement custom topologies.

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 in order 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.

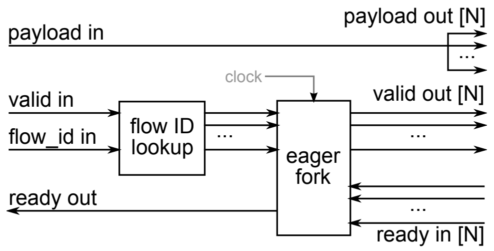

**Figure 3: Split Node architecture**

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 contain 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[4] stage. The Eager Fork is a sub-structure 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[4].

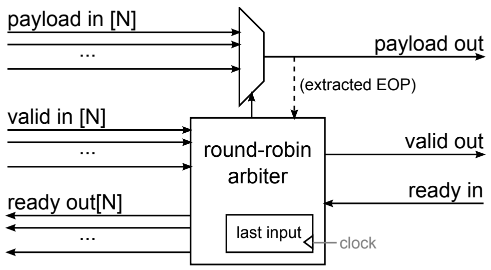

**Figure 4: Merge Node architecture**

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 multi-cycle 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).

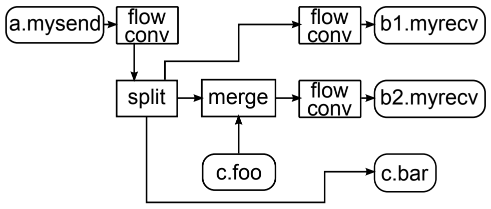

**Figure 5: Example System with Interconnect**

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 labelled 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. While 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 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'd like the interconnect, just like the block RAM being accessed, to have a determinstic 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 simutaneously 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, they 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 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 non-trivial 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 represents the System as a graph, with vertices representing Routed Streaming Interfaces belonging to both designer-specified functional modules and those of internally-generated interconnect modules. Vertices are labeled with their clock domain (which is fixed for designer-specified Interfaces and initially unknown for interconnect), and edges are weighted by their total payload widths. A greedy multi-way cut[7] algorithm is run on this graph to cut it into partitions representing clock domains, minimizing the total cut weight. Once the cut points are identified, clock converter FIFOs are inserted.

This feature 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 syntehsis tool. With this example, we also hope to better illustrate the nature of fine-grained interconnect and the challenges related to automatically synthesis.

Our application is a parallel LU Decomposition engine [16]. 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 (64x64) 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.

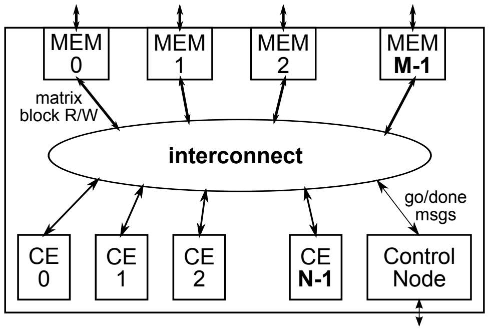

**Figure 6: LU Decomposition Engine**

The CEs, Memory Nodes, and Control Node are large (using between 1000 to 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 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 we have described so far, *and* to optimize across all those levels. However, in this paper we focus on the fine-grained system design within a *single* CE.

### 4.1 Compute Element 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.
- 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 which buffer it belongs to. 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 in order 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 since 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, while 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 require high-performance and non-trivial 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 – the actual connectivity between the Marshaller/Pipeline and the Caches is annotated in the figure, rather than being expanded out, for better readability. For all but two links (Pipeline to Top Cache reads), some mix of one-to-many or many-to-one communications is needed,

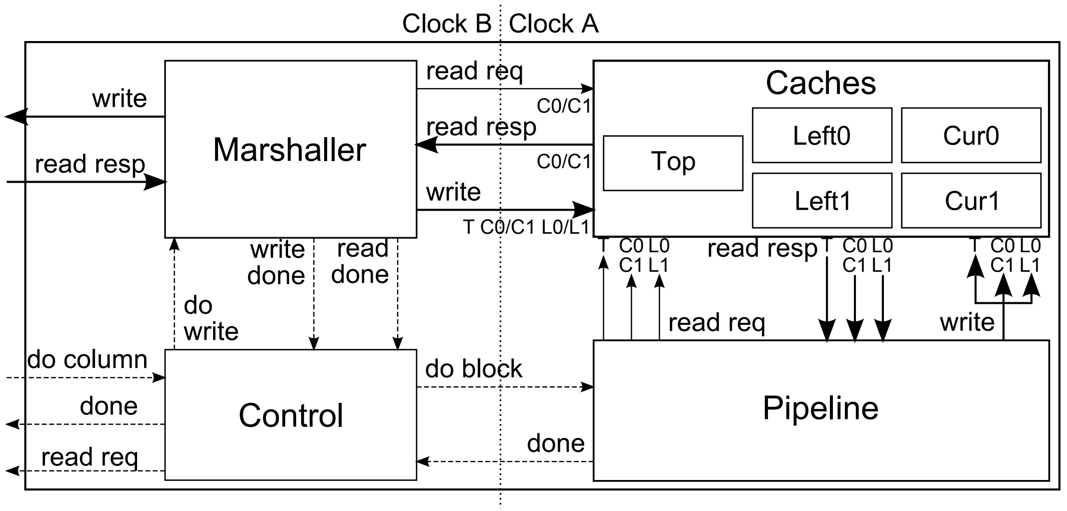

**Figure 7: Compute Element Architecture**

requiring distribution or arbitration hardware within the interconnect. Writes from the Pipeline also require broadcasting to multiple Caches during some modes of operation.

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, since 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 implementation 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[1] 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 in order 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, since they required the greatest interconnect complexity:

- Clock domain crossing
- Marshaller to Cache read path
- Pipeline to Cache read 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 travelling 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 9 FIFOs which 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. 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[2] 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 are 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 hand-made 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 GENIE variant handles the read path just as efficiently as the manual implementation, by virtue of allowing the system specification to declare that the Marshaller and Pipeline never compete for the same Cache ports. This generates Merge nodes that are nearly identical to the manual implementations, containing a mux which is controlled by the incoming Valid signals.

The Qsys interconnect has an arbiter block which is designed for the general worst case in which inputs compete for the output. However, even when there is no competition, we witnessed the generation of backpressure during the first cycle of a multi-cycle train of read requests. This required modifying the Pipeline by a FIFO in front of the read request port for the Current Cache, and another FIFO (wide, containing write data) in front of the write ports.

## 5. RESULTS

In this section, we quantitatively compare the three Compute Element variants presented in Section 4 in order 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. In this section 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.

## 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 TCL and Lua scripts, 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.

| | Interconnect + TOP | | FUNC | | TOTAL | |
|---|---|---|---|---|---|---|
| **Variant** | **Lines** | **Δ** | **Lines** | **Δ** | **Lines** | **Δ** |
| **Manual** | 1029 | 0 | 1323 | 0 | 2352 | 0 |
| **GENIE** | 290 | -72% | 1289 | -3% | 1579 | -33% |
| **Qsys** | 440 | -57% | 1411 | +7% | 1851 | -21% |

**Table 2: Code Line Counts - Designer Effort**

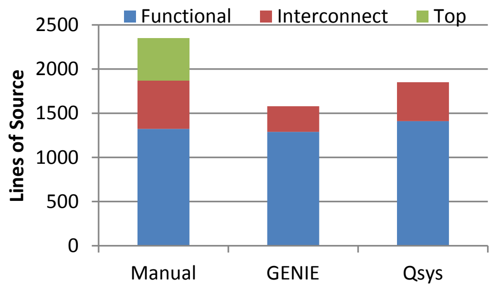

**Figure 8: Code Line Counts - Designer Effort**

Line counts were obtained using the CLOC[5] tool, which ignores comments and blank lines. Table 2 and Figure 8 give the results, with the table showing both absolute and relative (to the manual variant) line counts.

Note that the manual variant's 1029 lines of interconnect source code include 440 lines solely dedicated to the top-level Verilog module which instantiates all 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 8 gives it its own category to provide a better comparison of 'real' interconnect specification size. Nevertheless, using either system integration tool spares the designer from having to write the top-level instantiation code, so we include it in the "Interconnect + Top" code savings of 72% and 57% that GENIE and Qsys achieve, respectively.

The design of the functional modules is also affected by the choice of interconnect synthesis tool, in order to be compatible with protocols or mitigate lack of features, as described in Section 4.3. The Qsys variant required 7% more source code to make the changes described in Section 4.3. Meanwhile, the changes to the GENIE variant yielded a small savings, requiring no major architectural changes.

In the end, if a designer were to create the Compute Element with GENIE in mind from the very beginning, they would need to write 33% less source code than writing with no automation at all, with an even greater reduction if we focus on just the interconnect. 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 IOs rather than real device pins. Both clock domains in the design were over-constrained to 1 GHz, and results were geometrically averaged over 6 random seeds.

**Table 3: Clock Frequencies**

| Variant | Clock A | Clock B |
|---|---|---|
| **Manual (MHz)** | 406 | 461 |
| **GENIE (MHz)** | 400 | 500 |
| **Qsys (MHz)** | 320 | 394 |
| **GENIE vs. Manual** | -1% | +9% |
| **GENIE vs. Qsys** | +25% | +27% |

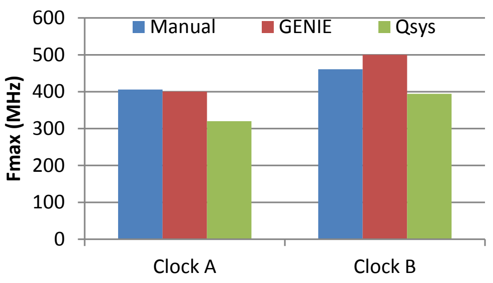

**Figure 9: Clock Frequencies**

Table 3 and Figure 9 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 1% slower than the manual variant's. In the Clock B domain, GENIE achieves a 9% frequency advantage, but at the cost of extra registers. Since our Stratix V device's frequency is limited to 450 MHz anyway, a design choice was made in the manual variant to use fewer register stages – a detailed level of control we hope to include in a later revision of GENIE.

Like the hand-made 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 4: Area Usage**

| Variant | ALM | M20K |
|---|---|---|
| **Manual** | 6739 | 40 |
| **GENIE** | 6987 | 40 |
| **Qsys** | 8383 | 92 |
| **GENIE vs. Manual** | +4% | +0% |
| **GENIE vs. Qsys** | -17% | -57% |

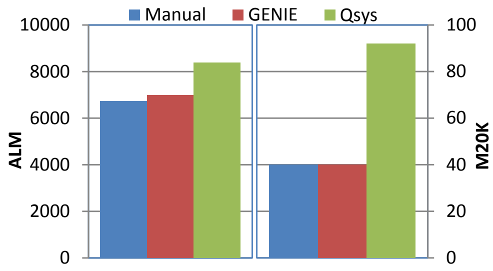

**Figure 10: Area Usage**

Table 4 and Figure 10 provide the area usage of the three variants, in terms of Stratix V Adaptive Logic Modules (representing logic, registers, and distributed memory) and M20K memory blocks. All variants also use 8 DSP (hard multiplier) blocks in addition to what is shown.

The GENIE-generated system is only 4% larger than the manually-created one, and occupies 17% fewer ALMs than the Qsys-generated system.

The Qsys interconnect contains an over-abundance 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 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 can't pack as many downstream pipeline registers as M20Ks can. These registers then go on to use additional ALMs.

## 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.

## 6. CONCLUSION

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 comercial 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 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 33% reduction in total source code line count compared to the hand-made implementation, and a significant 72% reduction if one only considers the code required to specify the interconnect. The cost for this productivity gain was a modest 1% decrease in achieved clock frequency (in one of the two clock domains) and a 4% 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 27% in the Compute Element's two clock domains, and a 17% 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.

Although this paper 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/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.

## 7. REFERENCES

[1] Altera Corporation. QSys - Altera's System Integration Tool. http://www.altera.com/products/software/quartus-ii/subscription-edition/qsys/qts-qsys.html.

[2] ARM Ltd. AMBA Open Specifications. http://www.arm.com/products/system-ip/amba/amba-open-specifications.php.

[3] L. Carloni, K. McMillan, and A. Sangiovanni-Vincentelli. Theory of Latency-insensitive Design. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 20(9):1059–1076, Sep 2001.

[4] J. Carmona, J. Cortadella, M. Kishinevsky, and A. Taubin. Elastic Circuits. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 28(10):1437–1455, Oct 2009.

[5] CLOC. CLOC – Count Lines of Code. http://cloc.sourceforge.net/.

[6] J. Cong, Y. Huang, and B. Yuan. Atree-based topology synthesis for on-chip network. In *Proceedings of the 2011 IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '11, pages 651–658, Washington, DC, USA, 2011. IEEE Computer Society.

[7] E. Dahlhaus, D. S. Johnson, C. H. Papadimitriou, P. D. Seymour, and M. Yannakakis. The Complexity of Multiway Cuts (Extended Abstract). In *Proceedings of the Twenty-fourth Annual ACM Symposium on Theory of Computing*, STOC '92, pages 241–251, New York, NY, USA, 1992. ACM.

[8] Y. Huan and A. DeHon. FPGA Optimized Packet-Switched NoC using Split and Merge Primitives. In *Field-Programmable Technology (FPT), 2012 International Conference on*, pages 47–52, Dec 2012.

[9] Lattice Semiconductor. LatticeMico System Development Tools. http://www.latticesemi.com/en/Products/DesignSoftwareAndIP/EmbeddedDesignSoftware/LatticeMicoSystem.aspx.

[10] A. P. Luca, L. P. Carloni, and A. L. Sangiovanni-vincentelli. Efficient Synthesis of Networks On Chip. In *in Proc. ICCD, 2003*, pages 146–150, 2003.

[11] U. Ogras and R. Marculescu. Energy- and performance-driven NoC communication architecture synthesis using a decomposition approach. In *Design, Automation and Test in Europe, 2005. Proceedings*, pages 352–357 Vol. 1, March 2005.

[12] M. K. Papamichael and J. C. Hoe. 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, pages 37–46, New York, NY, USA, 2012. ACM.

[13] PUC-Rio. The Programming Language Lua. http://www.lua.org/.

[14] V. Todorov, D. Mueller-Gritschneder, H. Reinig, and U. Schlichtmann. Deterministic Synthesis of Hybrid Application-Specific Network-on-Chip Topologies. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 33(10):1503–1516, Oct 2014.

[15] Xilinx Corporation. Accelerating Integration. http://www.xilinx.com/products/design-tools/vivado/integration/.

[16] W. Zhang, V. Betz, and J. Rose. Portable and Scalable FPGA-based Acceleration of a Direct Linear System Solver. *ACM Trans. Reconfigurable Technol. Syst.*, 5(1):6:1–6:26, Mar. 2012.