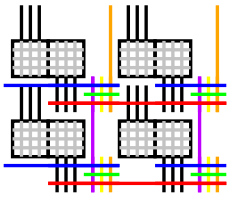


Implementation of Computation
Note 86: LIFT Design Notes

DJ Park, Yuanlong Xiao, David Glanzman, Hans Giesen, André DeHon

Last Update: 2017/11/28 01:35:49
Originally Written: November, 2017

This note is intended to elaborate design and design options for the LIFT overlay architecture introduced in [1].

Contents

1 Needs

Original rationale:

- link up separately compiled modules without the need to take time for monolithic place-and-route of large design
- allow incremental movement or replacement of leaf modules, again without the need for place-and-route time

To support that, we have some derived needs:

- Configure leaves – enable/disable, what they connect to, interface options
- Flow Control – deflection-routed network does not provide; need to prevent consumers from sending data the producer is not prepared to received or store
- Provide interface between Leaf processing and deflection-routed network

2 Clocking

Likely we will want network on separate clock from leaves. The division will be somewhere in interface to network.

We should be able to run the network at a relatively high fixed rate (400–800MHz). The network logic is fixed, so we can pipline for it, and possible provide placement directives for it. A high-throughput network will reduce the bottleneck the leaves see because we have a fixed, shared network.

The speed of the leaf nodes will vary and depend on how well designed they are. So, the leaf should not constrain the speed of communication.

Some things to address:

- Can leaves run at different speeds?
- How many different leaf speeds can we support? (e.g., do we need to provide a discrete set of clocks to all leaves and let the leaves pick which one to use?)

3 Wrapper Discipline

In order to accommodate a range of leaf demands, we will plan to use parameterizable wrappers to provide the interface between the compute block and the network. This can be parameterized for things like the kind of interface (streaming, read/write, bulk synchronous) and number of input and output of each type (Sec. 3.2). Since this logic changes with each compute leaf, we will put this interface logic in the physical (partially reconfigurable) leaf. Wrapper interface can provide pragma-controlled implementation options for autotuning (Sec. 3.3).

3.1 Self Node ID

Interrface logic will need to know its own node-id to know which packets to keep and which to resend because they were misrouted. We can configure the self-node-id as `#define` that gets defined during HLS compilation of leaf nodes once assigned to a leaf position in tree.

3.2 Parameters

- Number of input streams
- Number of ouptut streams
- ? stream width?
- Raw memory ports (Sec. 9)

3.3 Pragmas

These would not change functionality, but would impact implementation. These are things that can be autotuned. These can probably be ignored for a software implementation.

- Depth of buffers
- Full duplex implementation (Sec. 8.6)
- Window acknowledgement granularity
- *monitoring options* (Sec. 16)

3.4 Leaf Layered Architecture

The basic leaf architecture contains a number of standard interfaces as described in the following. Fig. 1 shows the general composition.

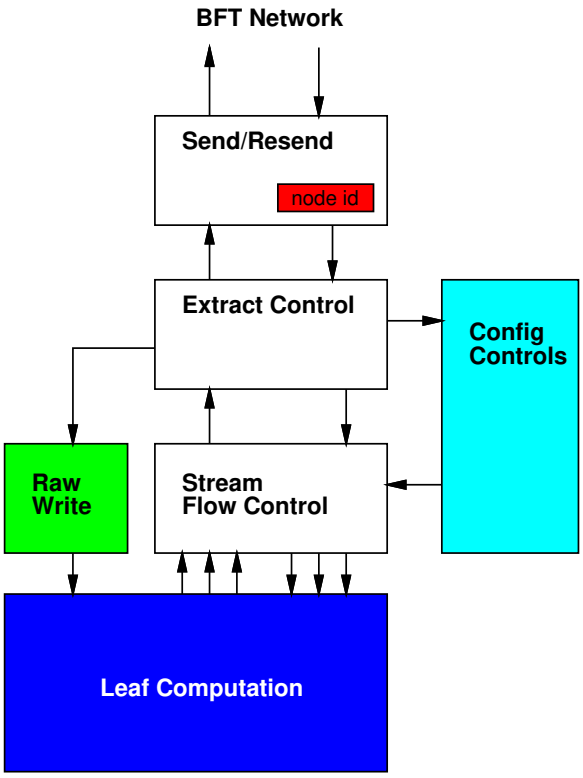


Figure 1: Leaf Architecture

4 Flit Format

Consistent with the deflection-routed design, LIFT uses single-flit packets. Our current thought is that the flit is 96b wide.

- node-id (8b)
- address (16b) – which is typically decomposed
  - port (4b)
  - address (12b)
- payload (64b)

The BFT only really cares that the top 8b are the node-id. The remaining 88b in the packet are uniterpretted by the BFT. It is up to the wrapper (Sec. 3) to interpret how the 16b of address are interpretted (e.g., split between port and address).

5 Deflection-Routed Tree Interface (Send/Resend)

The basic interface will be one input and one output port from network at each leaf. The leaf interface logic must deal with resending misrouted data back into the network, including its own network input that is deflected back to the leaf node.

Basic operation: if the incoming flit is not for this node send back to network. Otherwise, the leaf can inject one of own flits, if it has flits to send.

6 Leaf Control Interface (Extract Control)

The leaf will have a designated port for its own configuration. When data arrive on this port, the leaf interface will send to logic to control write into configuration registers.

We will need to configure destination for streams and other stream parameters (See Sec. 8.1). Similarly, we will likely need to configure location nodes and addresses for read/write interfaces (Sec. 9). Each node will also likely have an enable/disable configuration so that nodes can be disabled as part of reconfiguration (e.g. when being replaced, or maybe even when connected notes are being replaced).

7 Large Leaves

Our plan is to eventually support leaves that are multiples of the minimum leaf size. In the simplest case, this will combine the leaf area from multiple leaves without touching the network. That means these 2x-, 4x-, or 8x-leaves will have multiple physical BFT ports. Each such port will have its own node address based on the original 1x-leaf slots.

At a minimum, we could ignore the extra BFT ports, simply providing them with minimal logic so they do not inject packets into the network. More likely, we will want to be able to use the ports.

The next easiest thing to do might be to split the logical ports on the leaves among the physical ports available. A more sophisticated design might perform some internal switching or arbitration to select among multiple ports into the network. Since producers target a single port, it's not as easy to just take inputs from the network on multiple ports. If we knew both the producer and consumer were operating with multiple ports, we could split a stream across mulitple ports, but this would limit designs that shrunk one side of stream down to fewer ports.

In any case, the wrapper (Sec. 3) will need deal with the multiple ports.

Ideally, for many cases, the larger leaves will fall out of autotuning parameters within the leaf. This means it would be good to abstract the network ports from the leaf computation for the leaf designer so that the leaf will automatically exploit the additional ports as they become available.

8 Leaf Streaming Interface (Stream Flow Control)

Problems:

- need to deal with flow control; the deflection-routed BFT does not provide.
- need to deal with fact the network will not guarantee in-order delivery.
- need to be able to disable producer while reconfiguring (redirecting) consumer

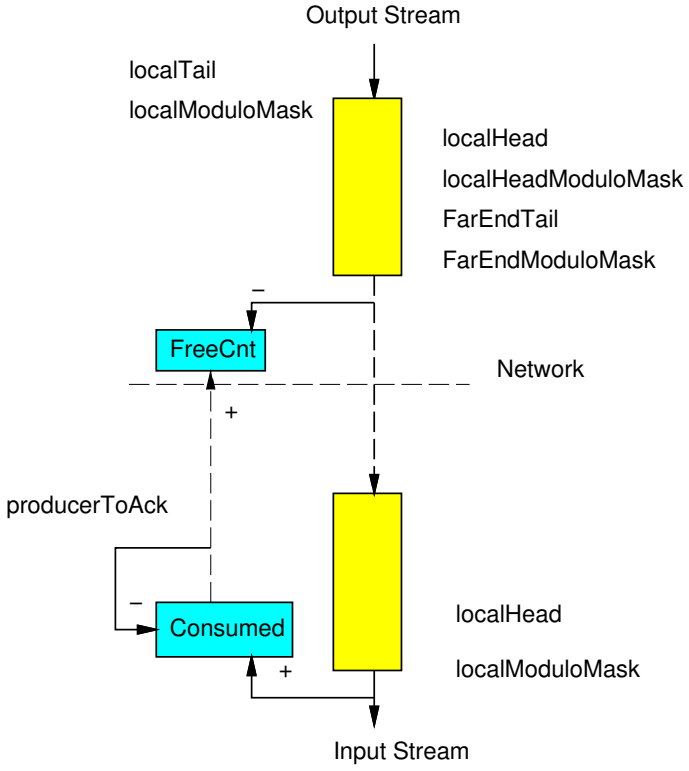
Also:

- want to be able to virtualize ports seen by processor

Strategy:

- Use Full-Empty bits
- FIFO is Ring Buffer on consumer side
- Keep track of space avaialble at consumer on producer side
- Windowed acks from producer to communicate when space available
- Producer side maintains Tail address for consumer side memory and flits write directly into consumer memory

The full-empty bits will guarantee that the consumer will stall for the next data iteam in the stream that has not arrived in order even if later items in the stream have arrived.



8.1 Control Registers Needed

Per output stream:

1. enable
2. FarEndTail (includes Node, port, address)
3. FarEndTailModuloMask
4. FreeCntr (possibly can infer from FarEndTailModuloMask, so maybe not need separate)

Per input stream:

1. producer to acknowledge (Node, port)

These registers need to be configured over the control interface to link up the leaf compute graph.

8.2 Producer Leaf Side

Writes into memory region for stream.

localTail pointer register keeps track of where to write into memory:

- Read localTail to check full/empty bit
- If empty, write into value with full/empty bit set to full and increment localTail
  - localTail=localTail & localTailModuloMask | (localTail+1)&not(localTailModuleMask)

8.3 Producer Network Side

Read from memory region for stream.

- If ((FreeCnt>0) and enabled)
  - Read localHead to check full/empty bit
  - If full
    - send value with FarEndTail
    - write empty bit on localHead
    - increment localHead (similar modulo increment)
    - increment FarEndTail (modulo increment)
    - decrement FreeCnt

Separately, upon receiving a WindowAck, add payload to FreeCnt.

8.4 Consumer Network Side

On each input flit:

- write into address with full bit set

8.5 Consumer Leaf Side

locaHead pointer register keeps track of where to read next:

- Read localHead to check full/empty bit
- If full
  - send value leaf
  - rewrite localHead with empty bit
  - increment localHead (modulo increment)
  - if increment will hit WindowAckTarget, send WindowAck and reset consumed to 0, otherwise increment consumed

8.6 Full Duplex Operation

One problem with this solution is that it demands a pair of reads and writes on each side in order to check and reset presence bits. That is, for each compute-side access it must read and write the presence bit. This means, it will need two ports for the compute side to achieve continuous operaiton. It also needs two ports on the network-send side. If we trust things are working properly, it can just write the data on network-receive side, so that side can tolerate a single port.

Since this is streaming access, we should be able to do better than a general 4-port memory.

8.6.1 Even-Odd Banking

One promising idea is to split the memory into even and odd banks. This way each the next read on any side is to the opposite bank of the writeback to clear/set the last presence value. Raw throughput-wise, this should work when virtualizing multiple streams onto a BRAM, so maybe this requires at least 2 BRAMs but can store many streams in them?

9 Leaf Remote (Slave) Read/Write Interface

The remote write interface is another personality to potentially support. At its simplest, it provides a raw write port into one or more memories that is available to the leaf computation. A more sophisticated verison might also include a read port. Generally, it could be parameters with separate selection of external read and external write capability on each memory. Also, perhaps, a separate configuration option for whether or not the memory port that it uses is shared with the compute leaf (i.e., the compute leaf uses that port when the interface is not using it).

This read/write interface can be used for initializing memory (e.g. instruction memory) and or reading back memory contents for debugging.

9.1 Parameters

Per port:

- read
- write
- share port
- write full (when a write from network, set full bit)
- maximum read burst length

9.2 Flit Formats

Write will have the same (generic) format we have been using (Sec. 4) with (node, port, address, payload). The read header (node, port, address) is the same, but the payload will have a specific format:

- read-burst-length (when parameter selected to support burst>1)
- reply-to-node
- reply-to-port
- reply-to-address

These directly become the header for the reply packet. The reply-to-node/port could, itself, be a write-port (or read/write port) interface.

Note that this format allows a single read (or write, or read/write) port to be used by many different masters.

For a port configured as both read and write, we will need to distinguish between a read and write request. An easy way to do this is to allocate a pair of port numbers and distinguish one port number as the read-request port and one as the write port.

9.3 Expression

How do we express these remote memory ports in HLS?

10 Leaf Initiator (Master) Read/Write Interface

If leaf nodes want to perform a remote read/write, we will need an output port type that will operate with the input Read/Write port interface (Sec. 9).

In the basic case, this is setup to read/write from a specific memory on a specific remote node (including the remote node associated with the DRAM, when appropriate). The port is configured with the remote node and port so the leaf simply issues a write a read with the remote address.

Reads demand that we specify where the response goes. If throughput isn't the goal, the leaf could stall for a response. A cleaner interface might be one that performs a remote fetch into a local memory, allowing the leaf computation to proceed. Cleanest might be one that uses full/empty bits as a scoreboard to allow split-phase read operations (or prefetch reads). This looks similar to a TAM split-phase read [2].

10.1 Scoreboard-remote-read

Remote fetch operation is: read remote address RA into local address LA (or read X words from remote address RA into local address LA).

- mark LA memory slot(s) to empty
- send remote read request to configured node, port for RA with associated node, port, and LA as reply node, port, address
- reply sets full bit during write
- node stalls if attempts to read an empty value from memory

10.2 Parameters

Per port:

- read
- write
- stall vs. raw-to-mem vs. scoreboard-remote-read
- maximum read burst length

10.3 Port Configuration

Per port:

- remote node, port
- associated write input port for response to read requests

10.4 Expression

How do we express these remote memory operations? Is is possible for them to appear in HLS as pointer arguments to compiled functions?

```
leaf_fun( uint64_t *in, uint64_t *out) {
    .
    .
    a=in[index]; // read stall
    local_in[li]=in[index]; // remote fetch to local memory
    out[index]=value; // remote write

    .
    .
 }
```

11 Graph-Machine Style Port Interfaces

For graph applications, we found it useful to have somewhat stylized edge communication interfaces [3].

As a start, we might use full/empty bits in memory with each input edge slot. We reset to empty between epochs (or when read from input slots) and write to full when written. Output edges need to track node, port, address for associated input edge. Output sends are a remote write to the associated node, port, address with the bit set to full. Wrapper support might include performing the output edge to node, port, address translation as a pipelined operation; that is, the leaf node writes to the output edge by edge id; hardware looks up the node, port, address in an output-edge memory and turns that into a remote write packet to be sent.

In a barrier-synchronized design, we can probably avoid the rewrite to set the presence bit to empty by making the presence bit an epoch bit (or bits) and consider it empty when the epoch bit in the memory doesn't match the current epoch.

For associative input edges, an unordered stream interface per node may be adequate. Receive inputs along an input port and process as arrive. If operating with barrier synchronization, this stream may not need flow control. This stream may also be different in that there are multiple producers feeding into this one stream. Here, output edges specify node and port for receiver. Since this is associative, we do not need the ordering, so can forgoe the ordering addresses and can stick into a FIFO in arrival order.

12 Barriers

By default, we have not discussed the BFT natively supporting barriers.

One way to support would be to overlay with a pair of barrier streams. When a node arrives at a barrier, it posts notice to the barrier arrival stream and then reads from the barrier complete stream. This stream-barrier will demand round-trip to a central resource and sequentialization of barrier arrival and barrier complete messages

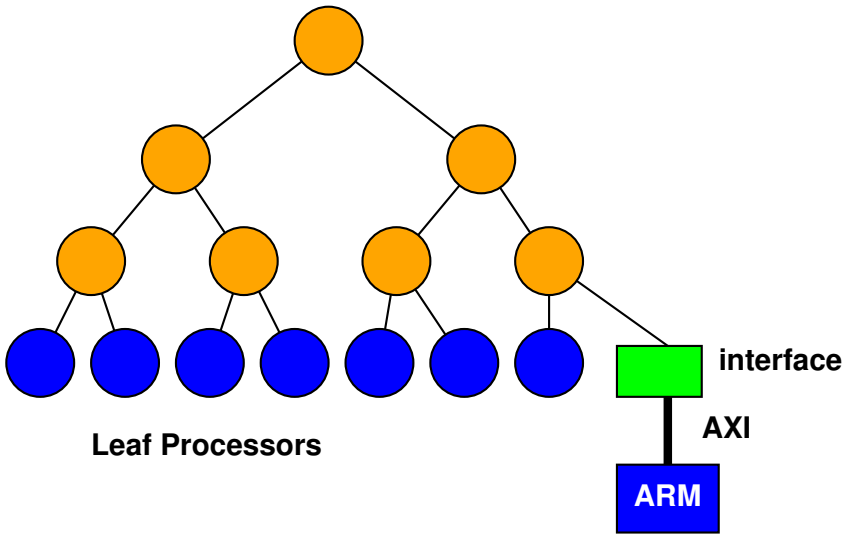
A barrier node (perhaps the control processor (Sec. 13) collects all the barrier arrivals and issues a barrier complete when it has a complete set. Barrier completion will need to send to all nodes.

The barrier arrival stream could be special with no flow control and simply counting messages on arrival, perhaps directly in hardware.

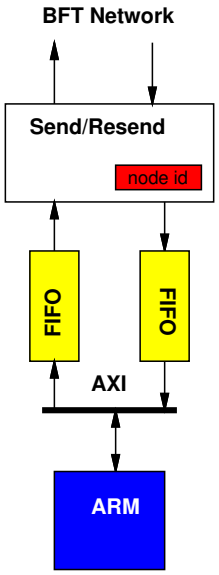
A barrier arrival reduce and complete broadcast tree will likely be higher performance at the expense of more virtual ports. This avoids bottleneck at a single barrier node for collecting arrivals and sending completions. It can be arrange by phyiscal locality to minimize network traffic.

13 Control Processor on Tree

To support configuration control and general interfacing, it will be useful to put a control processor on BFT. Typically, this control processor would be the ARM on Zynq. Specifically, we can use this to onfigure ports (and read/write BRAMs) from the control processor through BFT (See Sec. 6).



The simplest interface is the Send/Resend logic for interfacing with the BFT and a pair of FIFOs for input and output from the ARM processor. The FIFOs here deal with the clock boundary crossing between the ARM and the BFT, including dealing with making sure each send into the network is taken as a single flit. This is a raw interface with the ARM directly supplying the packet header information (node, port, address). In this simplest case there is no flow-control, so the program on the ARM will need to be careful not to overdrive the BFT network.



As we progress, we will likely want to build a more sophisticated interface for the ARM. This may add more of the leaf functionality including flow control and support for multiple streams. We will need to keep the raw interface (which specifies node, port, address) for configuration control (Sec. 6).

14 Also on Tree

We will also want to put other functions on the tree, including the DRAM and peripherals such as an ethernet MAC. These, too, will need some interface logic between the tree and the resource. The MAC might have streaming interfaces that can be configured to send to leaf stream input ports. The DRAM could have streaming channels to memory, block-transfer operations, and raw read-write operations.

TODO elaborate designs

15 Leaf Size Design Point

For large width (w=96), each switch should roughly be dominated by the datapath outputs. So, a T-switch will need about $3w$ LUTS and a π-switch will be about $4w$. For $p = 0.5$, per endpoint:

$$\left(\frac{\pi}{2} + \frac{2T}{4}\right)\left(1 + \frac{1}{2} + \frac{1}{4} + \ldots\right) \approx \pi + T$$

So, at $w = 96$, the BFT will require a little over 700 LUTs/leaf. Let's call it 750 LUTs.

If we put similar logic, another 750 LUTs, into the wrapper interface, that's about 1500 LUTs.

A clock domain on the Virtex UltraScale is about 10K LUTs. So, maybe we split this in half and get 5000 LUTs per leaf or 4250 for the leaf including wrapper, or 3500 LUTs for the leaf logic. User logic is 70% of the logic.

The MPSoC Zynq XCZU9EG has 274K LUTs, meaning we could fit about 54 leaf modules. The XCVU9P on the Amazon F1 has 1.2M LUTs, about 1M of which are available for user logic, suggesting around 200 leaf modules.

16 Monitoring Options

- count packets sent (received) per port
- count cycles resending misrouted packet at input port – combinded with packet count, this will allow us to characterize congestion at network ingress
- timestamp packets to measure latency (add to packet width?, fully record?, accumulate latencies?, just BFT latency or also in queues?)
  - lightweight state would be to add up the total latency of arriving packets; dividing that by total number of packets received will give us an average latency
  - also keep track of max latency and min latency
  - do we do this per BFT output or per port? per port is more state, but then tracks (at least in streaming case) per source-sink pair
- count misroutes in BFT at each subtree
- count traffic through subtrees in BFT
- *node/compute block latency...elaborate options* (heuristic of time form inputs to outputs? – sum up and take ratio to input count to estimate compute node latency?, max, min...)

References

[1] Dongjoon Park André DeHon, Hans Geisen. IC note 84: Introspection acceleration. Learning, Reconfigurable, Design-Space Exploration, Autotuning. <icnotes/84/ic84.pdf>.

[2] David E. Culler, Seth C. Goldstein, Klaus E. Schauser, and Thorsten von Eicken. TAM — a compiler controlled threaded abstract machine. *Journal of Parallel and Distributed Computing*, June 1993.

[3] Michael deLorimier, Nachiket Kapre, Nikil Mehta, and André DeHon. Spatial hardware implementation for sparse graph algorithms in GraphStep. *ACM Transactions on Autonomous and Adapative Systems*, 6(3):17:1–17:20, September 2011.