TyBEC Dataflow Scheduler and Buffer Generator

S. Waqar Nabi, June 2017

DFG Generated by TyBEC

TyBEC will parse TIR code and will internally create a DFG for the given design. asdf

Synchronous Domains, Asynchronous Gates, and the STALL signal

The DFG will be consist of one or more of *Synchronous Domains (SD).* This will be a collection of nodes that can be deterministically scheduled at compile-time with respect to each other. The scheduling is discussed later.

SD's will connect externally via specialized nodes called *Asynchronous Gates (A-gates).* In the simplest case, these A-gates will be communicating with the external DRAM (global memory). They can however also refer to channels talking to other devices in a multi-device scenario. In the context of TyBEC-generated designs, the communication between HDLs and their AOCL wrappers would be via A-gates, as these boundaries communications are always asyncrhonous and require handshaking.

All A-gates will be able to *stall* the entire SD anytime by asserting the STALL signal to all the nodes, e.g., in case where the input stream from a DRAM via an A-gate is not available.

Node Types

- Memory objects

- SSA variables

- autoindex

- arguments (ports)

- functions (inside hierarchical nodes)

- smache (for offset buffer creation).

Edge Types

Edges will map to streams.

- Explicit: using streamread/streamwrite

- Implicit: using SSA instructions

FIFO/Syncrhonization Buffers

Some edges will require buffers for syncrhonization. They are of two types:

- Explicit: using ALLOCA instructios

- Implicit: Inferred on edges (streams) for synch

Node Parameters

For the purpose of scheduling, every *node* in a TyBEC DFG graph will be desribed by these parameters[1]:

LAT = latency (input to output).

LFI = Local Firing Interval: How soon can subsequent executions on this node be fired, based ONLY on internal constraints (e..g if the node has ¿1 latency and is not i internally pipelined, then we have to wait for LAT before firing it again, even if data is available at its input.

EFI = External Firing interval: The firing interval determined by how frequently data ia made available by predecessor(s).

AFI = Actual Firing Interval: This will look at both LFI and the constraints of firing due to availability of data at input (i.e. AFI_p x FPO_p)

FPO = Firings per output

SD = Starting delay (based on synch-domain counter)

[1]Each node will also have parameters for the purpose of *costing*; they will be discussed elsewhere.

Scheduling

With the nodes as described by a five-parameter tuple, i.e. (LAT, LFI, EFI, AFI, FPO, SD), scheduling means calculating all these parameters for all nodes. They will be calculated as follows, where (TIR) means the paramter is derived directly from the TIR, so is known when the scheduling algorithm is run.

If there are multiple predecessors, then we calculate for all of them, and choose the maximum value.

For leaf nodes

All non-function nodes are leaf nodes. E.g. args, primitive instructions like add, mul etc, and inferred nodes like smache.

$LAT_{not.reduction} = latencyOfPrimitiveInstruction$

This latency of a primitive instruction would be available via a cost model (e.g. 1 for integer addition, 5 for floating point addition).

However, if the primitive instruction is a reduction operation, then the latency has to take into account the size of the array, and also the way in which the reduction is carried out:

$LATl_{linear.reduction} = latencyOfPrimitiveInstruction + SizeOfReduction - 1$

$LAT_{tree.reduction} = latencyOfPrimitiveInstruction + ...$

$LFI = (TIR)$

$EFI = AFI_p \times FPO_p$

$AFI = max(LFI, EFI)$

$FPO = (TIR)$

$SD_{inputNode} = SD_{parentFunction}$

$SD_{nonInputNode} = SD_p + (LAT_p - 1) + (AFI_p \times FPO_p)$

Since all output streams from a function have to be instep for scheduling at the next hierarchical level to work, we need to make sure all output nodes have same starting delay. The $SD_{nonInputNode}$shown above may lead to output nodes that are out of step. So, for output nodes, we need to set the SD to the maximim or all output nodes.

$SD_{node} = max(SD_{allOutputNodes})$

The SD of **all** output nodes in a function must be

For heirarchical nodes (functions)

This refers to both leaf functions[2]. We assume that input nodes and terminal nodes are symmetrical. If not, then we need to re-consider (TODO).

$LAT = SD_{outputNode} + LAT_{outputNode}$

$LFI = LFI_{inputNode}$

$EFI = AFI_p \times FPO_p$

$AFI = max(LFI, EFI)$

$FPO = FPO_{outputNode}$

$SD = SD_p + (LAT_p - 1) + (AFI_p \times FPO_p)$

TODO: How are these expressions effected when we split/merge?

For MAIN (Asychronous Gates)

Global Variables (alloca in addrspace = 1)

Main is the only function that talks to global memories (for now), and is the boundary for the IR design, so it needs to be treated slighltly differently fron other functions. The key thing to note here is that access to a variable in the global memory implies an **Asynchronous Gate**, and thus it does not require deterministuc scheduling like the other nodes. With the implicit understanding that there will be a handshaking protocol to deal with the asyncrhonity at these gates, for the purpose of scheduling we can then assume that these nodes are always ready to *fire,* have no latency, and start at time 0.

$LAT = 0$

$LFI = 1$

$EFI = 1$

$AFI = 1$

$FPO = 1$

$SD = 0$

Edge FIFO Buffers Sizes

FIFO buffers will be required on the edges for synchronization in a number of different cases.

[2]Note: leaf function is not the same as leaf node.

Case 1 - Mismatched latenies

For inferring buffers when the only syncrhonization requirement is due to different latencies of predecessor nodes, leading to different starting-delays on incoming edges, the SD is chosen to me the maximum one, and all other paths will require a buffer of this size:

$fifodepth_{edgeA} = SD_{longest-path-on-input-edges} - SD_{edgeA}$

Case 2 - Mismatched rates between producer and consumer

When the producer is producing faster than the consumer can consume (i.e. $EFI_p < LFI_p$) , then one approach is to stall the producer. But here we assume that we do not want to stall the producer, so we have to introduce a buffer that looks at the size of the entire stream, and then infers a buffer size that makes sure there is enough buffer space in view of the rate differences.[3]

$fifodepth_{edgeA} = \frac{size}{EFI_p} - \frac{size}{LFI_A}$

where $size$ refers to the total size of the stream.

TODO: Synchronizaing multiple predecessors producing at different rates.

Case 3 - Combination of maps and folds (mismatched FPOs)

If there is a map and fold on parallel paths (as is the case in the HIPEAC illustration), then we need to infer a buffer on the map path. In this case, the fold node will have an FPO that is greater than the FPO of the map path which would be 1. In such case, we simply infer a buffer on the map path as follows:

$fifodepth_{edgeMap} = FPO_{fold} - FPO_{map}$

This should be a general expression for the case when multiple predecessors to a node have different FPOs, and not necessarily limited to this particular illustration of map and fold on a parallel path.

Case 4 - Combination of cases - TODO

Also TODO: Verify all these expressions.

[3] If this turns out to be too large for the FPGA resources, then our cost-model should tell us, in which case we fall back to stalling the pipeline.