Automatic Generation of Translators for Packet-Based and Emerging Protocols

Brian Crafton[1], Arijit Raychowdhury[1], and Sung-Kyu Lim[1]

[1]Georgia Institute of Technology, Atlanta, GA
[1]School of Electrical and Computer Engineering
brian.crafton@gatech.edu

Abstract—**A recent trend in open source hardware and chiplet-based IP reuse faces a key obstacle: protocol standardization. Hardware interfaces lack flexibility and require designers to follow a strict behavior when implementing IP. The rigid nature of hardware interfaces prevents IP reuse, a critical challenge in integrating a plethora of emerging open source IP. To mitigate these challenges, we propose a tool to automatically synthesize translators between arbitrary IP blocks. Using a protocol description language (PDL), we model protocols such that they can be interpreted as finite state machines (FSM). Next, we design algorithms to map and schedule transactions between these protocols, generating a single integrated state machine which serves as a translator between the two protocols. Lastly, we convert our integrated state machine into readable RTL (Verilog) and perform functional verification. Our flow has been implemented, tested, and proven on 12 protocol pairs with unique behavior.**

I. INTRODUCTION

Very large scale integration (VLSI) of various functional components has enabled tremendous progress in modern computing systems. However, in the last decade the cost of designing a system-on-chip (SoC) has increased dramatically [1], motivating more efficient design principles. Fortunately, growing interest in open source hardware [2] coupled with emerging technologies like 2.5D silicon interposer [3] and multi-chip modules [4] promise to significantly reduce the cost of designing an SoC. These new technologies seek to enable drastically different design methodologies for SoCs, where IP blocks are *chipletized* and fabricated IP can be reused across several designs by mounting it in a system-in-package (SiP). Recent demonstration of chiplet-based designs have been fabricated with 64 and 96 processors [5], [6]. Although these designs feature only processors, it is expected that future work will include other functionality used in mobile SoCs like wireless communication and analog IP.

Meanwhile, FPGAs have recently become available in cloud computing applications for their ability to provide near ASIC performance without the cost of a custom ASIC. FPGAs are also commercially available at low cost with high quality embedded CPUs and mature software tools. With frameworks like SiP and FPGA in place, new design methodologies envision IP catalogs with hundreds of vendor chiplets to choose from that can easily be integrated into a large scale SoC. This new framework promises affordable high quality chip design without redesigning IP that can be completed in weeks rather than years. This IP can be provided either commercially or from emerging open source hardware IPs [7], [8], [9].

Although these new technologies and design flows can greatly reduce the cost and design time of modern SoCs, they face their own unique challenges that are avoided by traditional design flows [10]. The key obstacle we face in integrated these IPs is communication. Recent work [10], has attempted to tackle this problem at the physical level by automating the generation of I/O cells. This is of particular importance to low cost and quick design of the envisioned framework since the design time saved by design reuse is lost to custom design of I/O cells. Just like the physical level, we observe these same issues at the logical or protocol level. Throughout the years many open source and proprietary protocols have been developed, each designed to be a generic protocol to standardize communication. However, with emerging challenges and design needs new protocols are adopted and old IP has to be updated. If the promise of simple SoC design and integration is to succeed, we require a solution to protocol standardization.

In this work, we propose the use of a new protocol description language (PDL) and synthesis algorithms to automatically generate translators between IPs that communicate using different protocols. Originally protocol synthesis was proposed [11], [12] as an early high level synthesis (HLS) technique to promote design reuse. In fact, the original motivation was the belief that standardization of protocols would be impractical [13]. Building upon these works, we expand on both the modeling techniques and synthesis algorithms to enable protocol translator synthesis for modern bus and packet-based protocols. We present a visualization of our proposed tool flow in Figure 1. We begin with source PDL files modeling the protocols, which are then parsed and converted into abstract
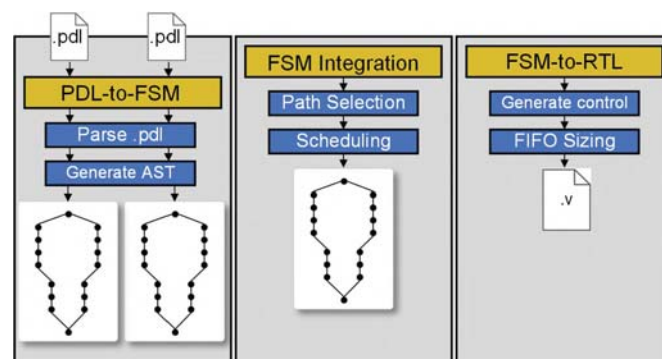

Fig. 1. A visual representation of the PDL-to-RTL flow. Initially, state machines are generated from PDL files, then merged into a single state machine, and finally RTL (Verilog) is generated.

syntax trees (AST) and later converted to FSMs. We then integrate the FSMs from both protocols by identifying paths in each protocol that send data and finding corresponding paths in the opposite protocol to receive the data. Next, we schedule the transactions and generate FSMs that model a valid translation between the two protocols. Lastly, RTL (Verilog) is generated using the integrated FSM as the control unit, and instantiating buffers and multiplexers to control data flow.

We use our tool flow to demonstrate the efficacy of automatic translator synthesis by testing on handpicked packet-based and emerging protocols requiring complex language semantics. We generate RTL and perform synthesis place and route for 12 protocol combinations, dramatically reducing RTL design and verification time. Furthermore we demonstrate methods to reduce both area and latency using automatic translator synthesis. We observe up to 39% area reduction from custom partial translator implementation.

II. BACKGROUND

Protocols and interfaces used between logic blocks can be modeled as finite state machines (FSM) [12]. The FSM modeling the protocol is a directed acyclic graph containing both vertices and edges. Vertices represent the current state of the system that is executing the protocol. Edges contain two essential components to model the protocol: conditions and transactions. Conditions are requirements for a state transition to be made, such as valid or ready signals. Transactions are the actual data to be transferred in the protocol.

In Figure 2, we illustrate this idea for a subset of the commonly used bus protocol: AXI [14]. In this example, we observe a value of *4* for *arlen* and asserted valid and ready signals. In the AXI protocol, this implies the master is attempting to read a sequence of four words from a memory. Upon satisfying these three conditions, the protocol proceeds to the next state in the FSM. Where upon continued assertion of valid and ready, the master protocol receives four transactions. Naturally we can model this behavior as an FSM, where these signals are conditions dictating state transitions, and the data transferred is mapped to the edges of the FSM.
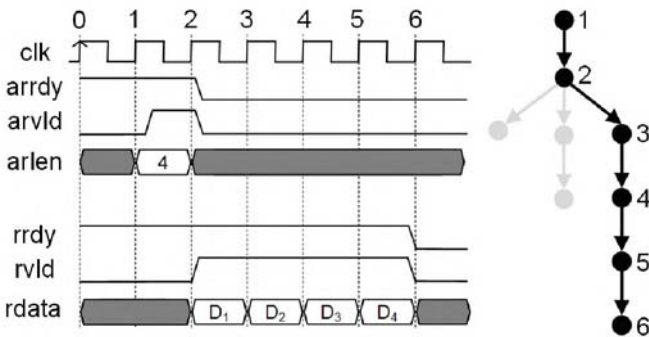

Fig. 2. Transformation of AXI into an FSM. Only three of the many branches of AXI are shown. The highlighted branch in the FSM is the path that was taken in the waveform.

Prior work [12] use a language (like HDL) so that designers can describe their protocols in a concise manner. This style of language is called protocol description language (PDL), and is used specifically to derive FSMs from protocols. Such a language yields similar benefits as HDL, promoting both optimization of the underlying data structure and designer productivity in describing it. Given two PDLs, and thus two FSMs, [12] showed that a translator between the two protocols could be generated by identifying and combining legal combinations of states between these protocols (e.g. one protocol sends data, one receives data). The result is a merged FSM, which could in turn be converted to HDL such as Verilog.

III. MODELING COMPLEX PROTOCOLS

As we discussed in the previous section, Logic-level protocols implemented by IP blocks can be modeled as FSMs. However, as protocols increase in complexity, additional features are required to adaquetly model their behavior. In this Section, we identify and describe components of popular bus protocols like [14], [15], [16] and packet-based protocols like [17] that require extensions to previously discussed protocol modeling efforts.

A. Packet/Flit Format

Packet based protocols are commonly used in multi-core processors and network-on-chips [18]. Packets differ from traditional bus protocols in that the physical wires do not carry the same meaning each cycle. Instead control signals and opcodes dictate what groups of wires carry a certain piece of information. To properly model this, the data must be virtualized from the physical wires that carry it. This is in stark contrast to bus protocols where the data and physical wires are the same entity.

Packets are usually divided over many flits where control signals necessary to decode data in one flit, were sent several flits before. This type of data structure, yields a tree-like FSM where each packet type has its own path in the FSM. In Figure 3, we illustrate a pair of example packets and their corresponding FSM. Although the tree-like structure of packet protocols requires slightly additional run time and state space, it does not result in exceptionally high area (Table II).

B. Independent State Machines

Modern bus protocols [14], [15], [16] allow reads and writes to occur at the same time, completely independent of each other. For example, the AXI protocol has 5 different interfaces to handle read and write transactions. While some pairs of these interfaces can be modeled in the same FSM, to capture the full behavior of the protocol, multiple state machines are required. Although in theory it would be possible to create a single state machine for all possible combinations of these independent state transitions, it is not feasible in practice given that the state space would increase exponentially. Therefore we propose that protocols should be a combination of many state machines that work independently of one another.
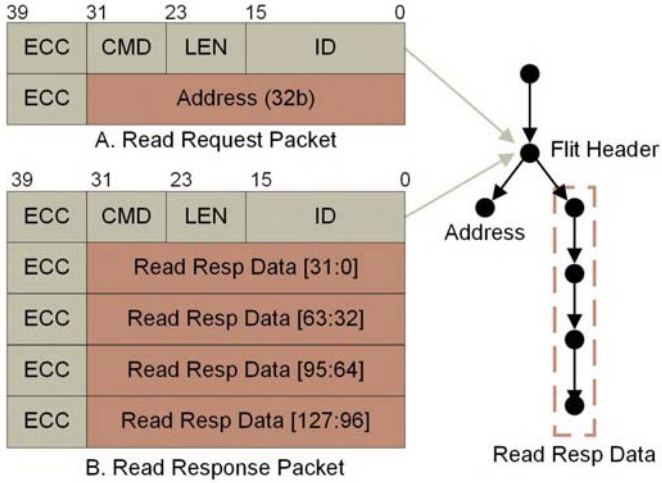
Fig. 3. Transformation of a packet protocol into an FSM. Both packets structure is defined by the header, after which they diverge.

C. Data Dependence

Although we require independent state machines to model modern protocols, it is possible that these state machines depend on one another. In AHB, for both a write transaction and a read transaction, the address must occur before the data is sent. During a burst mode transaction, the subsequent addresses and data can occur on the same cycle or different cycles. The only requirement is that each address must precede the data it corresponds to. In Figure 4, we show an ideal scenario where the data follows the address and the sequence finishes after 5 cycles. In this case data dependence is not enforced since the address always precedes data. However, since AHB allows for stalls, we must support sequences like Figure 5. The sequence in Figure 5 effectively breaks the single FSM model for AXI because address and data are stalled and do not follow each other. To properly model this behavior we require multiple FSMs and information regarding this relationship between *waddr* and *wdata*. More specifically, we must be able to declare dependence between these two variables, and enforce the condition that address precedes the data it corresponds to.
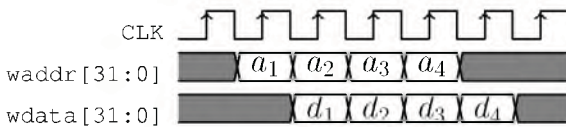

Fig. 4. The "easy" case for data dependence, no stalls occur and data always trails address by 1 cycle.

IV. PROTOCOL DESCRIPTION LANGUAGE

We have developed a new language that draws on similar syntax to the Verilog hardware description language. Although useful for modeling hardware, both Verilog and System Verilog provide insufficient details to properly model a protocol.
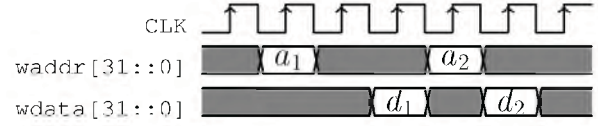

Fig. 5. A "hard" case for data dependence. Stalls occur for both data and address and data does not always follow address by 1 cycle.

The goal of PDL is to fully describe a protocol in the most concise and simple way possible. The PDL should effectively model the interface of the protocol and the necessary behavior of that protocol for translation. It is this behavior that HDLs do not model. Rather than arbitrary ports, the tool needs to know what rules the protocol follows.

Prior works have proposed two necessary definitions to synthesize a protocol [12], [4]. The first is a description of the signals that implement the protocol. In defining these signals, the designer must describe the name, width, direction, and type. The second is a description of the behavior of the protocol that can be modeled with FSMs. The edges within the FSM contain the conditions necessary for a state transition, and information about the data that is transferred during the transition. While this model suffices for smaller protocols, modern protocols require additional language semantics to express complex behavior. In this section we describe the necessary semantics, and later demonstrate how they can be used for bus protocols and packet protocols.

The additional semantics we include in our language are as follows:

1) *Transaction Id*: "Id" gives a transaction an identifier so that other transactions can reference it to declare dependence. This allows us to determine when stalls are required in a sequence (Figure 4).
2) *Transaction Dependence*: "Dep" allows a user to specify that the given transaction is dependent on another transaction. With this identifier we can enforce order in the protocol.
3) *Address Offset*: "Offset" describes the size of the data an address refers to.
4) *Data Range*: "Range" specifies the amount of data (in bits) that is sent during a transaction.

Each of these parameters is simple, yet adds necessary information to properly model modern protocols. In this work we focus our effort on bus protocols and packet-based protocols. The bus protocols we have discussed in this work all require multiple state machines and the ability to describe data dependence. Meanwhile, packet protocols require a structured hierarchy describing how previous flits determine future data to come. In the following subsections, we further describe these requirements and illustrate how these behaviors can be modeled using simple code.

A. Bus Protocols

The bus protocols we have discussed in this work all require multiple state machines and the ability to describe

```
protocol axi
  ports
    data     out 32 araddr
    control out 8  arlen
    control out 1  arvalid
    control in  1  arready
    ...
    data     in  32 rdata
    control in  1  rvalid
    control out 1  rready
    ...
  endports
  behavior
    generate tid = arlen:0 begin
    + @(rvalid(1), rready(1)) {
      rec( rdata(range=32, dep=araddr(tid)) )
      }
    end
  endbehavior
  ...
  behavior
    generate tid = arlen : 0 begin
    + @(arvalid(1), arready(1)) {
      send( araddr(offset=4, id=tid) )
      }
    end
  endbehavior
  ...
endprotocol
```

Fig. 6. A Subset of the AXI protocol PDL for read transactions.

data dependence. As an example, we show a subset of the AXI protocol for read transactions in Figure 6. Both the read address and read response portions of the protocol are described in separate state machines. We use generate loops for all possible burst transactions while declaring dependence between address and data.

B. Packet Based Protocols

Using the same data structures and language, it is possible to define a packet based protocol. Although additional semantics are required, the resulting data structures are processed the same way. A packet based protocol uses multiple flits, a packet structure, and a variable length bus to transmit data. A packet header will define an opcode or command and the length of the packet. These control signals will determine the rest of the packet structure and what transactions will occur. Just like the control signals in regular bus protocols, we create branches based on these control signals because it will determine the rest of data to come.

Given that most packet protocols are proprietary, we opt to define a generic packet protocol featuring common transactions like reads and writes. In Figure 7, we show a subset of this protocol (also illustrated in Figure 3) for read transactions that we will later use to generate a translator for AXI. Each packet is described as the concatenation of several sub-packets, including a header and body. We use sub-packets to reduce redundant code in the protocol. We use a branch based on *CMD* field to infer which type of packet is being sent. There are over a hundred different packet commands for this protocol, but we only show *ReadResp16*. This packet is six flits long. This includes the header, the address information, and then four packets containing ECC and write data.

V. FSM INTEGRATION

Once we have parsed both PDL source files and have generated the FSM data structures, we have the underlying model and can ignore whether a protocol was a bus or packet-based protocol. Instead, we simply view a protocol as a set of FSMs, where each FSM is a graph of states and edges like the example provided in Section III. With the two sets of FSMs, the goal of FSM integration becomes creating a single set of FSMs that serve as a translator between both protocols. More specifically, we must ensure that all transactions sent by one protocol are received by the other.

In [12], an algorithm was first proposed to automatically synthesize a translator between two protocols. Their algorithm attempts to recursively traverse the two protocols modeled as FSMs. At each state, a check for data consistency is performed. If the data sequence sent by a protocol is correctly matched by the receiving protocol, then the combined sequence of paths is appended to the combined FSM. If one protocol attempts to send or receive while the other does not, then data sequencing is incorrect and the path is deemed invalid.

This exhaustive search algorithm was shown to perform poorly on more complex protocols [19]. Later work [19], [20], used a divide and conquer approach to more efficiently search the solution space. However, we still find this heuristic to be unnecessarily complex. Instead of traversing the whole FSM, we take a more abstract approach. Our proposed algorithm considers the data transactions that each path in the FSM contain, and then only attempts to match other paths that contain the same transactions. Next, we attempt to schedule the paths by issuing stalls when there is a timing mismatch or the protocols have different bus widths. We further detail these steps in the following subsections.

```
packet packet_header begin
  control 6 LEN
  control 8 CMD
  control 8 TID
  ...
  control 5 ECC
end
packet rd_rsp_pkt (i, d) begin
  data 32 read_resp_data(range=32, id=i, dep=d)
  control 3 CD
  control 5 ECC
end
packet pkt_protocol begin
  subpacket pkt_header pkt_header0
  begin
    ...
    | + @(pkt_header0.CMD(0)) : RdReq16
        rd_req_pkt
    | + @(pkt_header0.CMD(1)) : RdResp16
        control 4 SID
        control 1 H
        data 32 rd_address(offset=16)
        control 3 ECC
        subpacket rd_rsp_pkt rd_rsp0(0, RdReq16(0))
        subpacket rd_rsp_pkt rd_rsp1(1, RdReq16(0))
        subpacket rd_rsp_pkt rd_rsp2(2, RdReq16(0))
        subpacket rd_rsp_pkt rd_rsp3(3, RdReq16(0))
    ...
  end
end
```

Fig. 7. A Subset of the packet protocol PDL for read transactions.

A. Path Selection

In any directed acyclic graph there exists some number of paths from source to sink, where each path is a unique set of edges. The set of all paths is all the possible procedures that a protocol can make. To create a functional translator, we must ensure that all data sent from both protocols is received by the other. To do this, we iterate through the set of all paths in the target protocol that transmit data. Then, we identify paths in the other protocol that receive the data the target protocol is trying to send. Once we satisfy all paths that send data, our translator is complete.

To demonstrate this, we show an example case our tool has successfully executed. In the process of integrating the packet protocol and AXI, we must find a translation for the *ReadResponse16B* (shown in Figure 3) packet of our packet-based protocol. In Table I, we visualize the data structure we use to represent the paths. Our tool identifies receiving paths that can be used to satisfy the *ReadResponse16B* packet. We also maintain information on the number of cycles and how many additional states the receiving path will incur. This allows the algorithm to make decisions based on performance and area. In this particular example, there are four paths in AXI that could satisfy *ReadResponse16B*. However, for paths 1 to 3 we would need to iterate several times. Choosing path 4 results in the lowest latency because the 2 cycle overhead is only incurred once. For paths 1-3 we would need to incur this latency at least twice, and in the case of using path 1 four times, we incur this latency four times.

However, there is a trade off to consider. Path 4 may require the fewest cycles, but it also requires 2 additional states in our FSM, while paths 1 and 2 do not add additional states since they have already been added to satisfy smaller read packets. Our tool defaults to always choose the lowest latency given that additional states are inexpensive, however, if area is a concern a flag enables area minimization. We provide psuedo-code for this in Algorithm 1. The search time for this algorithm is O(N) since we iterate through all the paths in protocol 1, but use a lookup table to identify paths in protocol 2 that satisfy the translation. This algorithm also takes a cost function as input argument to decide which path is the best solution for translation. This cost function is a weighted function specified by the user to optimize for either area or latency.

B. Scheduling

In the path selection part of the flow, we are not concerned with timing, rather just making sure we obey the constraints set on each path. After we find the set of paths from the second protocol to match with our first protocol we must

---

Algorithm 1 Path Selection

```
1: procedure PATH SELECTION(P1, P2, Cost)
2:     for  path1 ∈ P1.paths  do
3:         for  path2 ∈ P2.paths.contain(path1)  do
4:             champ = min(Cost(path2), Cost(champ))
```

Table I. Path Selection

Path #	#Cycles	Path Used	#States	Data Transfer
1	3	Y	0	(4B rdata, 4B addr)
2	4	Y	0	(8B rdata, 8B addr)
3	5	N	1	(12B rdata, 12B addr)
4	6	N	2	(16B rdata, 16B addr)

create a schedule that issues the control signals and satisfies the timing constraints of each protocol. The basis of our scheduling algorithm is simply checking if data has been sent by master, and if data is ready to be received by the slave. We provide psuedo-code for this algorithm in Algorithm 2. Although simple for cases when both protocols send data with the same width, it is challenging to handle data of different sizes. For instance, if *IP1* communicates using 32-bit AXI and *IP2* communicates with 64-bit AHB, we must buffer packets going to *IP2* and split packets sent to *IP1*.

This issue is further exasperated by serial protocols and packet based protocols. For the same scenario above a serial *IP1* would require 64 buffering cycles before sending a single transaction to *IP2*. Furthermore such a translator would potentially require asynchronous FIFOs for serial protocols that operate faster than bus protocols. Packet protocols suffer from a different issue. Since the data we send and wires we send it on are not bound, but virtualized, the same data transaction can come from different physical sources. Our scheduling algorithm accounts for this by adding control signals to the multiplexer preceding the FIFO mapped to the data transaction. In our FSM, we add this control signal based on our current state so that we select the correct data to buffer.

Algorithm 2 Schedule

```
1: procedure SCHEDULE(master, slave, fifo)
2:     while  !master.empty() and !slave.empty()  do
3:         if  !fifo.full()  then
4:             fifo.push = master.send
5:         if  slave.receives()  then
6:             slave.receive = fifo.pop
```

VI. RTL GENERATION

Once an integrated FSM is generated, it needs to be converted into a synthesizable RTL where it serves as a translator between IP blocks in an SoC. Given the ease of modeling state machines in HDL and the availability and maturity of commercial grade synthesis tools, we choose to convert our integrated FSM into Verilog code. Like [21], we find that the repetitive nature of translators makes it feasible to generate Verilog using higher level blocks such as FIFOs, multiplexers, and adders. Furthermore, the excellent optimization provided by these tools minimizes area, further simplifying the task of generating high quality translators.

In each translator, the integrated state machine directly serves as the control unit. This control unit makes state transitions based on inputs from the connected IPs, and issues control logic to multiplexers, FIFOs, and address calculators

to control the flow of data. Multiplexers are instantiated throughput the design to route the data from one IP to the FIFOs and address calculators, and then to the other IP. This is especially important for packet based protocols where several sources share the same destination. This occurs in packets when data resides in different parts of the packet or when different bus widths are used.

To buffer data transactions between IPs, we use register based FIFOs inside the translator. In the FSM each state contains FIFO and mux control logic, which can be implemented as a lookup table indexed by a state variable. However, because we generate Verilog and use commercial synthesis tools we allow the tool to find a more optimal implementation. To handle addresses sent between protocols, we use a generic address calculator block. This is necessary to translate between protocols that receive data in different sizes. For example, AHB receives bursts up to 16 4 byte transactions while AXI sends bursts up to 1024 transactions. In this case, we must break the 1024 transactions into 64 bursts. For each burst we must provide an additional address which is the original address plus an offset.

A. Control Generation

With our control unit and basic blocks in place, the next step is generating the control logic. The control logic from the IPs to this control unit (input control) is what we used in path selection and scheduling to satisfy our constraints. These signals are already embedded in our translator and we use them to set the conditions for state transitions. The control logic we seek to generate is from the translator to the IP (output control) and from the translator to the multiplexers, FIFOs, and address calculators. All of this control logic will be generated for each state in our control unit. Physically, this can be implemented as a lookup table indexed by our state variable. However, because we generate Verilog and use a commercial synthesis tool, we allow the tool to find a more optimal implementation.

We first consider how to generate the output control logic to the IPs. To do this, we iterate through all the states and edges and generate output control for each edge based on conditions and transactions from that edge and the two states it connects. Given that the PDL already specifies what conditions lead to state transitions, we can embed these values for the case when we do not wish to stall. However, if we do wish to stall based on the current state of the protocol we must generate control logic to do so. Since we have the full specification of the protocol it is possible to compute a set of control signals such that the protocol stalls. For most protocols this is a simply inverting the valid or ready signal. In the case of protocols that do not stall one a transaction is initiated, we must have a sufficiently sized buffer to buffer a full transaction sequence. This is further discussed in the following subsection.

To generate the control signals for the basic blocks in the translator we follow a similar procedure. We iterate through all the states and edges and generate the necessary signals based on whether data is sent or received. We then append these signals to the set of signals to be stored in the lookup table of the control unit. For each FIFO, we must control two signals: push and pop. For each edge, we simply check if data is sent or received and then we set push and pop accordingly. For each address calculator, we control the base address and the offset of the current edge from the start of the transaction sequence. In operation, this value is then added to the base address in the address calculator block. For each multiplexer, we must control the select signal. For each edge we check the source of the input data and simply set the select value to the port mapped to our input data source.

B. Automatic FIFO Allocation

To ensure sufficient buffer capacity and minimize translator area, we automatically size our FIFOs using information from our FSMs. Given that we have complete information about the protocol and the transactions along the different paths of the protocol, we can compute the minimum FIFO depth required to buffer each transaction along a single path. One solution is to choose the total number of bytes sent along the longest path. However this option will not minimize FIFO area. A better solution is to traverse the entire the FSM, and simply choose the most bytes buffered at one time. This implies that by default we can only buffer a single transaction sequence and that one sequence must finish before another can begin. Additional FIFO capacity can be added so that more than one sequence of transactions can be buffered, however by default we assume a sequence is finished before beginning another.

C. Verilog Generation

After generating necessary control signals and appropriately sizing buffers, we have a sufficient design specification to write out the Verilog for our translator. First we trivially write out the ports in Verilog syntax using the port information specified in the PDL. Next we instantiate the multiplexers, FIFOs, and address calculators from template Verilog implementations. Lastly, we write out our control unit in Verilog syntax. This is implemented as a large case statement, where each state is modeled by a case where output and basic block control logic is specified.

Figure 8 shows a block diagram of an AXI to packet protocol translator. In this case FIFOs are connected to all data ports with multiplexers to handle routing into the packet IP. We group all control signals together since they all feed into the control unit. In this design we represent all four state machines in our translator as the single control unit. This control unit takes all control signals from both IPs and issues control logic back to the IPs as well as the multiplexers, FIFOs, and address offset blocks. The architecture for translators is repetitive, thus automating the RTL generation using high level blocks simplifies the problem space.

VII. RESULTS

We have run our tool on several different protocol combinations including AXI [14], AHB [15], TileLink [16], and our generic packet protocol. The examples we have chosen are to demonstrate the unique aspects of protocols that our tool
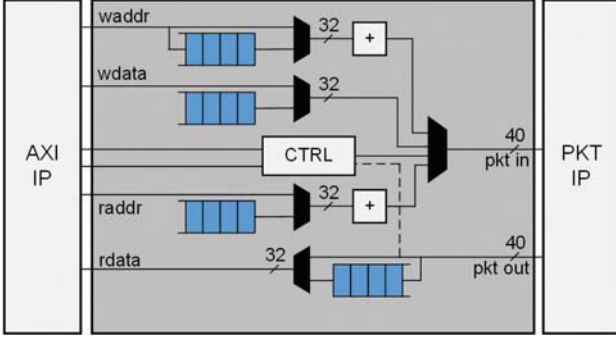
Fig. 8. RTL Block diagram for translator between AXI and our packet protocol. All logic to and from *CTRL* unit is control logic.

Fig. 9. Layouts of two translator designs in 28nm technology node. (A) AXI to Packet (B) Packet to AXI

Table II. Translator Synthesis Results

Translator	State	Time (s)	Cells	FF	Footprint (μm^2)
AHB AXI	214	0.06	2017	619	2393
AHB PKT	261	0.04	1888	551	2222
AHB TL	242	0.04	1999	611	2313
AXI AHB	1079	0.78	2195	623	2496
AXI PKT	1184	1.10	2318	591	2491
AXI TL	1322	1.56	2634	682	2568
PKT AHB	375	0.08	1971	547	2212
PKT AXI	392	0.16	1966	550	2228
PKT TL	380	0.16	1849	515	2024
TL AHB	914	0.24	2193	600	2393
TL AXI	989	0.33	2355	606	2494
TL PKT	1061	0.33	2612	541	2228

can handle. Each example is synthesized and implemented in 28nm through our automated back-end flow. This flow uses our compiler to generate translator RTL and then performs synthesis and place and route using commercial tools. In Table II we show the synthesis and place and route results for each translator. These results from this backend flow provide us with valuable feedback to ensure that the area of translators remains small.

In Table II, we also include the runtime for our tool and the number of states in the control unit. Naturally we observe that the states and runtime are proportional to the total area of the design. The translators including AXI or TileLink (TL) protocols have the highest area and consequently, the most states and highest runtime. Despite a fairly large state space, all the translators we create take roughly a second to synthesize. It should we noted that for the larger translators, we use multiple threads to explore the set of paths, so *CPU time* will be greater than *wall-clock time*. In comparison, modern synthesis and HLS tools take days of runtime on small parts of large CPUs and GPUs since it becomes infeasible to synthesize them as a single component. Thus as more complex protocols emerge, we expect that translator synthesis runtime will remain insignificant compared to gate level synthesis.

As we have observed so far, automatic protocol synthesis presents an opportunity to both reduce design time optimize
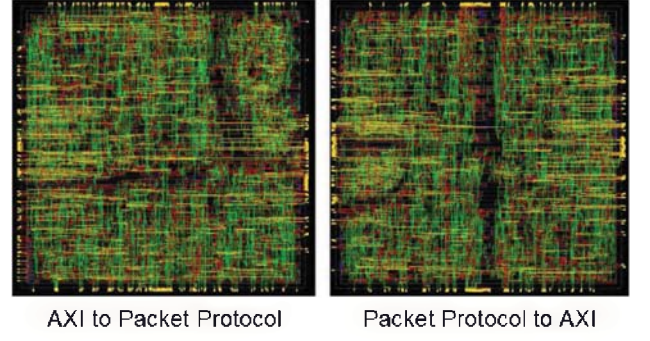
for area. In the following subsections, we discuss these two improvements separately.

A. Design Time Reduction

Although we have to manually write the PDL code, it is needless to say that it takes a small fraction of the time it takes to write and verify optimized RTL for 12 translators. Furthermore, we realize that once a verified PDL file is written, it need not be written again. Hence, the four protocols we demonstrate in this work become an asset of our tool so that any additional protocol added can be synthesized with existing PDL. This implies that potential design time saved grows quadratically as we increase the number of translators in our PDL library and the maturity of our tool. This attribute becomes a critical asset in our tool's application in future design flows. Where vendors can provide PDL for each IP and then integration tools can automatically generate custom optimized translators for the SoC or SiP.

B. Area Reduction

In Section I, we discussed two methods that can be used to optimize for area in the protocol translator problem. The first of these is partial translator synthesis. This idea stems from the fact that most designs rarely (if ever) use the full functionality of the protocol, but they still incur the area cost of a full translator between each IP for this unused functionality. To demonstrate how PDL can reduce area by simplifying the customization of protocols, we have removed some of the less commonly used transactions in each protocol's PDL and rerun our flow. In Table III we show the partial translator results from this experiment. The area of each translator reduces by roughly 30% as shown in Figure 10, further increasing the area savings automatic translator synthesis enables for large scale SoC design.

VIII. CONCLUSION

In this paper we proposed several contributions to the protocol translator problem that we implemented in over 15 thousand lines of C++ code. We proposed requirements to model packet based and emerging protocols and the language semantics to properly express them. We demonstrated an efficient path selection and scheduling algorithm to create the

Table III. Partial Translator Synthesis Results

| Translator	State	Time (s)	Cells	FF	Footprint (μm^2)
AHB AXI	119	0.06	1242	363	1842
AHB PKT	207	0.03	1154	329	1410
AHB TL	112	0.03	1200	332	1698
AXI AHB	638	0.31	1420	367	1715
AXI PKT	743	0.35	1222	272	1513
AXI TL	832	0.68	1517	386	1860
PKT AHB	207	0.07	1220	351	1422
PKT AXI	224	0.15	1372	399	1554
PKT TL	220	0.15	1369	374	1548
TL AHB	607	0.21	1357	351	1645
TL AXI	719	0.29	1444	370	1787
TL PKT	783	0.23	1319	312	1515
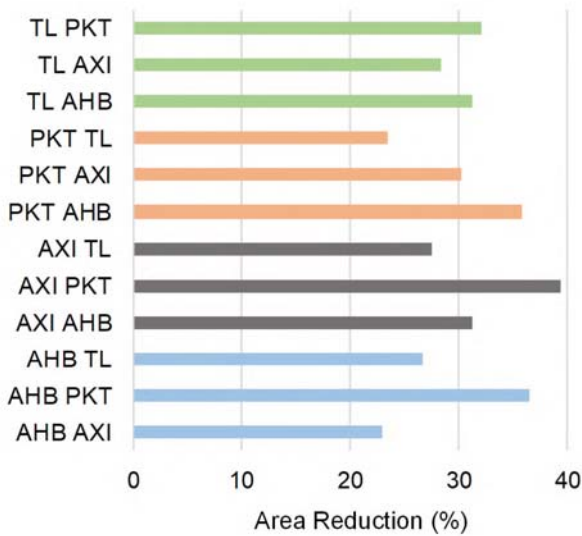
Fig. 10. Bar plot of area reduction from full to partial translator implementation

control unit for our translator, which uses configurable cost function to minimize latency and area. Next, we demonstrated a method of taking the data and address transactions and mapping them to FIFOs and address calculators, and then routing them together using networks of multiplexers. From our implementation we generated the RTL and performed synthesis and place and route for 12 protocol combinations. Using automatic translator synthesis we observe up to 39% area reduction from partial translator implementation, and we greatly reduce the design time required for integrating many IPs in an SoC.

REFERENCES

[1] G. Gielen, P. De Wit, E. Maricau. J. Loeckx, J. Martin-Martinez, B. Kaczer, G. Groeseneken, R. Rodriguez, and M. Nafria. "Emerging yield and reliability challenges in nanometer cmos technologies," in *Proceedings of the conference on Design, automation and test in Europe*, pp. 1322–1327, 2008.

[2] K. Asanović and D. A. Patterson, "Instruction sets should be free: The case for risc-v." *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146*, 2014.

[3] J. Kim, G. Murali, H. Park, E. Qin, H. Kwon, V. Chaitanya, K. Chekuri, N. Dasari, A. Singh, M. Lee, *et al.*, "Architecture, chip, and package co-design flow for 2.5 d ic design enabling heterogeneous ip reuse," in *Proceedings of the 56th Annual Design Automation Conference 2019*, pp. 1–6, 2019.

[4] T. Fukushima, T. Konno, K. Kiyoyama, M. Murugesan, K. Sato, W.-C. Jeong, Y. Ohara, A. Noriki, S. Kanno, Y. Kaiho, *et al.*, "New heterogeneous multi-chip module integration technology using self-assembly method," in *2008 IEEE International Electron Devices Meeting*, pp. 1–4, IEEE, 2008.

[5] P. Vivet, E. Guthmuller, Y. Thonnart, G. Pillonnet, G. Moritz, I. Miro-Panadès, C. Fuguet, J. Durupt, C. Bernard, D. Varreau, *et al.*, "2.3 a 220gops 96-core processor with 6 chiplets 3d-stacked on an active interposer offering 0.6 ns/mm latency, 3tb/s/mm 2 inter-chiplet interconnects and 156mw/mm 2@ 82%-peak-efficiency dc-dc converters," in *2020 IEEE International Solid-State Circuits Conference-(ISSCC)*, pp. 46–48, IEEE, 2020.

[6] S. Naffziger, K. Lepak, M. Paraschou, and M. Subramony, "2.2 amd chiplet architecture for high-performance server and desktop products," in *2020 IEEE International Solid-State Circuits Conference-(ISSCC)*, pp. 44–45, IEEE, 2020.

[7] K. Asanovic, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, *et al.*, "The rocket chip generator," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*. 2016.

[8] R. Balasubramanian, V. Gangadhar, Z. Guo, C.-H. Ho, C. Joseph, J. Menon, M. P. Drumond, R. Paul, S. Prasad, P. Valathol, *et al.*, "Miaow-an open source rtl implementation of a gpgpu," in *2015 IEEE Symposium in Low-Power and High-Speed Chips (COOL CHIPS XVIII)*, pp. 1–3, IEEE, 2015.

[9] F. Farshchi, Q. Huang, and H. Yun, "Integrating nvidia deep learning accelerator (nvdla) with risc-v soc on firesim," *arXiv preprint arXiv:1903.06495*, 2019.

[10] M. Lee, A. Singh, H. M. Torun, J. Kim, S. K. Lim, M. Swaminathan, and S. Mukhopadhyay, "Automated i/o library generation for interposer-based system-in-package integration of multiple heterogeneous dies," *IEEE Transactions on Components, Packaging and Manufacturing Technology*, 2019.

[11] J. Akella and K. McMillan, "Synthesizing converters between finite state protocols," in *Computer Design: VLSI in Computers and Processors, 1991. ICCD'91. Proceedings, 1991 IEEE International Conference on*, pp. 410–413, IEEE, 1991.

[12] R. Passerone, J. A. Rowson, and A. Sangiovanni-Vincentelli, "Automatic synthesis of interfaces between incompatible protocols," in *Proceedings of the 35th annual Design Automation Conference*, pp. 8–13, ACM, 1998.

[13] G. Borriello, "Specification and synthesis of interface logic," in *High-Level VLSI Synthesis*, pp. 153–176, Springer, 1991.

[14] A. AMBA, "Protocol specification v2. 0," *ARM Holdings plc Std*, 2010.

[15] T. AMBA, "Specification (ahb)(rev 2.0)," *ARM Ltd, May*, vol. 1, p. 1, 1999.

[16] H. Cook, W. Terpstra, and Y. Lee, "Diplomatic design patterns: A tilelink case study," in *1st Workshop on Computer Architecture Research with RISC-V*, 2017.

[17] G. Casey and S. S. C. Team, "Gen-z an overview and use case s," in *Open Fabric Alliance, 13th annual workshop*, 2017.

[18] W. J. Dally and B. Towles, "Route packets, not wires: on-chip interconnection networks," in *Proceedings of the 38th annual Design Automation Conference*, pp. 684–689, 2001.

[19] S. Watanabe, K. Seto, Y. Ishikawa, S. Komatsu, and M. Fujita, "Protocol transducer synthesis using divide and conquer approach," in *Design Automation Conference, 2007. ASP-DAC'07. Asia and South Pacific*, pp. 280–285, IEEE, 2007.

[20] M. Fujita, H. Tanida, F. Gao, T. Nishihara, and T. Matsumoto, "Synthesis and formal verification of on-chip protocol transducers through decomposed specification," in *Quality Electronic Design (ISQED), 2010 11th International Symposium on*, pp. 515–523, IEEE, 2010.

[21] A. Grasset, F. Rousseau, and A. A. Jerraya, "Automatic generation of component wrappers by composition of hardware library elements starting from communication service specification," in *Rapid System Prototyping, 2005.(RSP 2005). The 16th IEEE International Workshop on*, pp. 47–53, IEEE, 2005.