HPR System Integrator: A SoC/Multiblade Link Editor

David J Greaves
Computer Laboratory, University of Cambridge, UK.
Working draft, Dec 2017.

***Abstract*—The IP-XACT XML schema allows digital electronic subsystems to be described in terms of their bus interfaces and configurable parameters. It provides basic support both for physical wiring (net-level) interconnection between components and for transactional method calling between component models. But in order to dynamically select pre-compiled subsystems for instantiation by an augmented high-level synthesis (HLS) flow, additional information about these components needs to be recorded. Missing details include the presence of combinational paths, deadlocking or conflicting sequences of operations, power-up sequences, whether the component contains internal state and basic timing parameters, such as the expected processing latency and required re-initiation interval. Some child IP blocks may also require access to centralised memory resources and management/debug infrastructure. The layout of their data in shared memories needs orchestrating to avoid clashes, as in software link editing. Child IP blocks may also require other blocks to support them in an overall dependency tree.**

In this article we give an overview of an open-source tool chain, based on augmented IP-XACT, that supports instantiation of pre-compiled blocks and black-box components. These components can be used either as FUs in an augmented HLS compiler (eg. for custom arithmetic) or for system-on-chip or system-on-multichip automated assembly. The tool automatically selects, configures and wires up the required supporting blocks. The result is synthesisable RTL that can be loaded into an FPGA (or multi-FPGA platform) for scientific acceleration as well as a SystemC high-level model and other report and visualisation files.

Our contribution is extensions to IP-XACT for community adoption.

I. Introduction

The advantages of modular design are well established. As well as non-functional advantages, such as revison control, design reuse, specification, testibility and so on, there is a potential performance advantage if more of the design space can be explored in the available time when deploying existing modules compared with a monolithic compilation.

High-level synthesis (HLS) provides a means to rapidly generate an RTL design. As is well known, such tools mostly operate by choosing a set of *functional units* (FUs), such as ALUs and RAMs, binding the operators, variables and arrays present in the high-level program to these resources and constructing a static schedule that represents the original behaviour. The schedule may be manifested by a controlling finite-state machine (FSM) that embeds the control flow of the high-level program, or sometimes a pipelined implementation is created that accepts streaming data every n-th clock cycle under external stream orchestration. Much of the synthesis tool's time is consumed by exploring alternative designs, in terms of fine-grained schedules.

The approach of a monolithic synthesis run does not scale to very large designs. It is unsuited to multi-FPGA designs. It does not support rapid recompilation of just part of a large project.

In this article we describe a method for incremental compilation for the domain of accellerators for scientific computation on FPGA. An *incremental* approach enables the results from a number of individual compilations to be combined, as with software libraries and link editing. In hardware, existing components are often parametrisable in terms of bus width or FIFO depth. Moreover, there are various combinations of components that can achieve the same functional result, whereas with software there is far less flexibility. We present a tool flow that enables manual and automatic selection of components from libraries and where incremental compilation results can be added into the library for future use. Each component consists mainly of a synthesisable RTL design file (or files) that is described by an electronic data sheet in extended IP-XACT format. There may also be SystemC or other high-level models of the component. For an FPGA platform, the baseline component library contains models and wrappers for the specific hardened resources available in that FPGA family. This is then augmented by importing IP blocks from third parties, manual RTL coding, augmented by sufficient meta information. or RTL generated from HLS runs.

Four specific advantages are:

1) **Compilation Speed**: Although logic synthesis and place-and-routet in the FPGA back-end tools is often the dominant contributor to design compile time, it is also important to have good compile time in a high-level synthesis tool. This is especially useful where the HLS tool can give performance and area predictions without needing to invoke the FPGA tools, or when the resulting RTL is to be tested on a small dataset using a fast RTL platform, such as Verilator (www.veripool.com). Incremental compilation, using an automated tool, such as `Make`, is always used wherever possible in complex system build. Only the parts directly affected by an edit need to be recompiled.
2) **Controller State Reuse**: A simple HLS tool will flatten inter-procedural control flow by expanding method bodies in-line. This can make the generated controller very big. Even though back-end logic synthesiser tools typically re-encode the resulting state

machine so that the output function is simple to decode, having more than a few thousand states breaks current-day tools. One possible solution is to give the controller a run-time stack, but the controller may remain centralised and its outputs may still not be generated close to where needed. On the other hand, where subsystems are synthesised separately, the controlling FSMs, where present, are intrinsically separate and their states are re-used for each invokation. Prime examples are trig and log functions, random number generation and I/O marshalling such as ASCII to/from floating point. When these components are stateless the tool can simply deploy as many instances as it likes, guided by metrics, with no growth in the parent controller's complexity.

3) **Parallelism**: Where a component is instantiated more than once, these can be used in parallel, providing a better performance.
4) **Spatial Awareness or Floorplanning**: A simple HLS tool may not be spatially aware. It is more sensible to take wiring length into account when making binding decisions. Wiring length is the critical factor in today's technology, whether for FPGA, ASIC or PCB. FPGA s have always had an abundance of flip-flops. Adder and multiplier density is also no longer generally an issue either (except for multiply-rich designs, such as CNNs, that are better addressed with dedicated synthesis tools rather than HLS [1]). In the future, thermal limits may increasingly affect FPGA design approaches. So for general HLS, having multiple instances of components, spatially localised, and perhaps less utilised, is a reasonable design point. This is especially so for multi-FPGA designs.

Component boundaries for incremental compilation need to be well chosen. Certain boundaries are enforced by FPGA size in multi-FPGA systems. Other boundaries arise from engineering concepts such as APIs, e.g. for the file system and maths libraries. Boundaries within user designs are potentially more flexible. When compiling an object-oriented high-level language, the object boundary is an intuitive cut point. Where constant values are passed as an argument to an FU, its implementation can often be simplified. For instance, multiplication by a constant can always be performed more quickly in hardware than multiplication by a variable (the number of adders needed is proportional to the number of bits different from their neighbour and is zero for a power of two). These differences change the schedule generated by an HLS compiler and the same level of optimisation cannot be achieved by mere inter-component constant propagation during back-end logic synthesis. Hence there are both penalties and benefits arising from an incremental compilation approach and the compilation unit must not be overly small. HLS tools typically have hardwired strength reduction rules that are applied when instantiating an FU, but for general incremental compilation, these rules need to be generalised for an imported FU library. For instance, `a+a` can be strength reduced to `a<<1`, replacing an adder with wiring, and `a*5` would be changed to `a+(a<<2)`.

Given that the RTL assembly for each FPGA will be presented to a monolithic run of the FPGA vendor's logic synthesiser, our tool flow can expect certain inter-block optimisations to be performed, such as propagating constant inputs into an off-the-shelf block and removing internal logic that computes unused outputs. We exploit this later in our automatic stitching of blocks using relatively heavyweight and standardised inter-block protocols on the basis that unused handshake nets and control logic will be deleted.

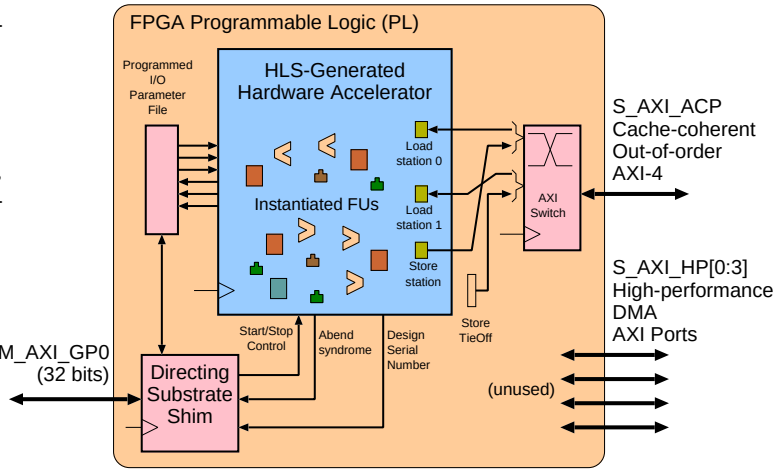

Fig. 1: Instantiation of a monolithic HLS result on Zynq-like platform. Programmed I/O serves the director port with an AXI slave and two in-order load ports and one store port are multiplexed onto an AXI-4 port for out-of-order service.

Figure 1 shows a traditional setup when a single HLS product (light blue) is instantiated in an FPGA such as Xilinx Zynq. Although the HLS compiler will have instantiated FUs, there are none whose connections range outside the HLS product. The product has a standardised control and management interface on the left-hand side. This supports low-performance host interaction for parameter setting and start/stop control using progammed I/O. High-bandwidth data movement uses the numerous load and store stations on the right that need connecting to main memory. The more stations it generates, the greater the number of outstanding transactions on main memory can be and the opportunities for out-of-order service are increased. With only a few ports, the system may be latency-limited and not able to exploit the raw bus or memory bandwidth, but this depends on the predictability of the appication's data access patterns as the tool unrolls loops. In the Figure, two stations are load-only and one is store-only. The data loaded and stored might be single words, but equally it could be bursts of consecutive locations. The multiplex of a number of these ports, implemented with the illustrated AXI switch, will have multiple outstanding transactions that can be served out-of-order provided replies are routed back to originators correctly. [1] For cache-consistent operation on Zynq, all of the traffic must be routed over the so-called ACP hard connection. Higher throughput is available by direct access to DRAM using other hardened AXI ports, but the lack of cache consistency may be a performance penalty where data is being shared on a fine-grain basis wit the Arm cores (not shown).

[1]The Kiwi HLS tool actually generates ports that are a little simpler than AXI-lite, but these are adapted to AXI-lite using automatically-deployed, lightweight bus adaptors that are not shown in the figure for clarity and the concentrating switch accepts AXI-lite on the left-hand side and delivers full AXI with transaction tags on the right.

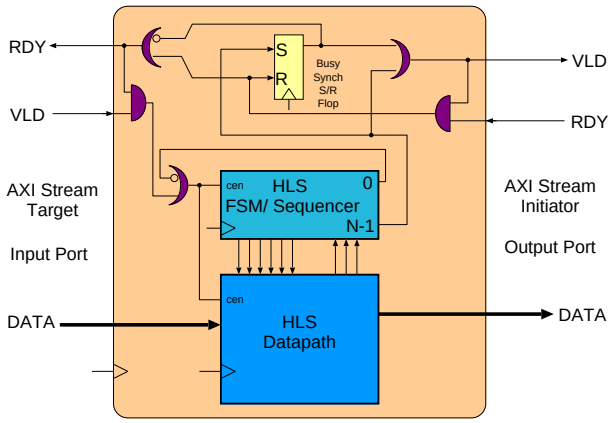

Fig. 2: A non-pipelined component wrapped as an AXI-streaming instantiatable component.

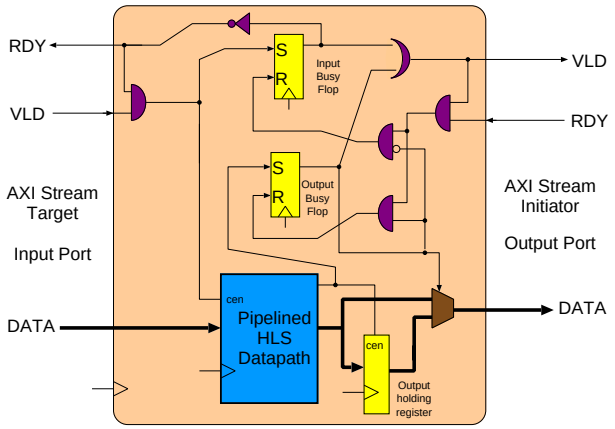

Fig. 3: A pipelined component wrapped for AXI-streaming with re-initiation interval of unity.

The available hardened resources vary accorss FPGA families. Some FPGAs have multiple hardened DMA controllers, such as AWS F2, which has four.

When an FU is used by a parent, a data transfer protocol is required. Handshaking nets are needed on any FU that is not fully-pipelined. A *fully-pipelined* component is apply to accept a new argument every clock cycle (*initiation interval* of unity) and delivers its result some pre-determined number of clock cycles later, known as its latency. For instance, block RAMs found in most FPGA families are available in fully-pipelined form with a read latency of unity. Power control protocols may also need to be exercised. Our HLS tool implements a number of handshake protocols. It will select an appropriate one and document it in an IP-XACT file written out at the end of block compilation. Figures 2 and 3 show how such child IP blocks are wrapped up to be used as AXI-streaming IP components. The associated handshake nets are placed around the main synthesis results that would be present for standalone operation. These are namely the datapath and the FSM sequencer, or just the datapath when it is pipelined. This is the difference between the two Figures. Two other paradigms commonly occur. One is where the input and output ports are combined and use a common pair of handshake nets, with the arguments being conveyed on the active edge of request and the result being returned on the active edge of the acknowledge. The other is for simplex sources or sinks that are 'always-ready'. For instance, reading a constant from a component simply needs that component to drive a dedicated output bus with an RTL literal tie-off. This needs no handshake at all. Similarly, a postable write that is always accepted needs only the write strobe direction of the handshake.

II. IP-XACT

IP-XACT is an XML schema allows digital electronic sub-systems to be described in terms of their bus interfaces and configurable parameters. It is an IEEE standard with two editions, the most recent in 2009 [2]. It was originally targeted for automated configuration and integration of assemblies of IP blocks and one might imagine it captures most of the information needed for this to be done as part of an HLS flow. However, it fails to document many of the fine details required for the optimised schedulling of operations. This is because it was intended mainly as the underlying representation for a GUI tool where port connections are made by hand. The standard has seen relatively wide adoption (e.g. in *Socrates* from ARM and *IP Integrator* from Xilinx) and it has been successful in its aims. It allows for, so-called, vendor-extensions, which are heavily used in practice. It is the purpose of this paper to kick off discussion leading to community adoption to address these issues.

The three forms of IP-XACT document we need to consider each represent one of:

1) a bus specification, giving its signals etc;
2) a leaf IP block data sheet with links to the design files;
3) an hierarchic component wiring diagram that describes a sub-system by connecting up or abstracting leaf components.

The bus specification (or abstraction as they call it) provides both for transactional modelling and net-level interconnection. It supports point-to-point port connections and broadcast connections. It describes whether a bus needs to be connected and the default tie-off logic values to apply when it is left disconnected.

Formal specifications of bus protocols are not described within IP-XACT. We do not address that in this paper either. In this work we assume that a number of standard, named, protocols are embedded in pre-existing blocks and synthesis tools and so the formal specifications do not need to be conveyed between compilation units. However, small variations over instances of protocols, such as the address and data bus widths and number of data lanes are reported in standard IP-XACT.

III. Incremental Compilation

Our approach to Incremental Compilation is to provide the infrastructure for a standard build system, such as `Make`, to be able to trigger re-compilations once a change has been made. We do not describe here any automation of the partitioning of the system into suitable-sized units or automatic creation of the Makefile. We currently rely on users inserting pragmas

into the application's high-level source code and creating their own Makefile. But these aspects can be automated in further work.

Our system maps method calls in the high-level source code to busses for arguments and results on the silicon. The busses typically have handshake nets of various forms, but certain interfaces do not require any (these being an always ready, read only port and the arguments and results for a fully-pipelined component). When a component is edited in its source form, the HLS tool is re-run on it. If its signature has changed, its parents may also need to be recompiled. If its silicon area has significantly increased, our System Integrator tool may need to be re-run, which, in the worst case, may move blocks between FPGAs, resulting in several FPGAs needing to be put through logic synthesis again.

Our approach directly supports passing scalar values between components over the busses. It also supports shared-memory between components. This enables object handles to be passed between components. Shared scalar global variables are not supported but can be mapped into a degenerate form of shared memory if desired. Our approach does not explicitly support a message-passing packet and queue system, but this is easily built on top in the high-level language using the shared memory underneath.

Figure 5 illustrates a typical structural set-up arising from multiple compilation units assembled on a single FPGA. In detail, the Figure shows a top-level application (primary IP block) that instantiates a separately-compiled child component that, in turn, instantiates three grandchildren of two different types. The children and grandchildren are subsidiary IP blocks. They do not do anything unless commanded by a primary IP block. Each compilation unit connects to its child by an arg/result port that is customised to the signature of the method being invoked. It is application-specific (A/S).

In addition, each child component requires access to RAM resources. In this particular example, the top-level module did not require RAM access (although it could well have its own BRAM privately instantiated).

Finally, every component has a directorate port for error reporting. The primary IP block also receives its run/stop control via this port.

In order to dynamically select pre-compiled subsystems for instantiation by an augmented high-level synthesis (HLS) tool, additional information needs to be recorded. This includes the presence of combinational paths, whether a component is mirrorable and certain timing parameters, such as its expected processing latency and re-initiation delay. The top two sections of Table I give the extensions included in the IP-XACT files generated by our HLS tool when it produces a component to be read in at a later stage of incremental compilation. The same tags can be added by hand to pre-existing library blocks from third parties or descriptions of hardened IP blocks from FPGA suppliers. We support more than one method call via a port. For instance, the port might support read, write and reset methods, with the address bus being shared for both the read and write methods.

The most important features of a port are already described in standardised IP-XACT. Whether a protocol is fixed or variable in latency, always ready, etc., is detected by the presence of standard attributes and logical net names in the bus abstraction document. For efficient synthesis we also need to know its expected latency. This will be precisely its actual latency if it is a pipelined, constant-latency implementation; otherwise it will be a nominal value to be used for static schedulling. When pipelined, the reinitialisation period may be greater than one: e.g new arguments presented every 3 clock cycles. When postable it is not necessary to wait for the response before re-issuing a request. A group of three broadly similar boolean attributes are cacheable, side-effecting and referentially transparent. A side-effecting method is treated as volatile and must be invoked even if any result is to be ignored. It must not be invoked specualtively. A cacheable method needs only be called once with a given argument. A referentially-transparent method will always return the same answer for a given argument and, if not side effecting, need not be invoked on a component where it has already been invoked on a mirror. A posted operation requires no response and so requests can be sent down a long channel without consequence in theory. However, posted operations are typically writes and sequential consistency must be observed [3]. An abortable method can be started without consequence, other than the abort dynamic energy being consumed. It is useful to know whether inputs must be held during a multi-cycle operation. Similarly, whether the outputs are held until they next need to change is useful. Both of these can save on temporary holding registers in the caller.

A given IP block that sports several method calls can potentially be invoked re-entrantly. However, hardware resources inside the implementation may not be sufficiently replicated. For instance, some of the same floating-point ALUs might be used. The HLS tool can be instructed over this sort of policy or may decide itself (we support both). Whichever is used must be reported so that the parent conforms.

When balancing combinational delays, the parent needs to know what paths exist through child components. We report this. When a port is bridged between FPGAs, it is possible for the combinational delay to go up and/or down. Such bridges tend to be synchronous with significant sequential delay but minimal combinational delay, so the final design can end up being conservative on critical path. The static timing analyser in the FPGA vendor tools is therefore unlikely to report a problem. Where many bus connections share a bridge, owing to instantiated concentrators, again the sequential delay may be increased.

IV. System Integrator Tool

As said already, child IP blocks may also typically require access to centralised memory resources and control and debug infrastructure. The layout of their data in shared memories needs orchestrating to avoid clashes, as in software link editing. Child IP blocks may also require other blocks to support them in an overall dependency tree structure.

To automate all of this assembly, we implemented a tool called the System Integrator. This tool connects up components and partitions logic between physical FPGAs. It instantiates protocol adaptors, concentrators and aggregators. It generates an hierarchic netlist to wire up the ports on all blocks using pre-

Property	Element Name	Abbreviated Description	Units
Per-Port attributes used for incremental compilation:			
Expected latency	`expected-latency`	How long between args and result.	clock cycles
Reinit period	`reinit-latency`	Minimum interval between new args being presented.	clock cycles
Postable	`postable`	Whether a response needs collecting.	boolean
Cacheable	`reftans`	For a method, whether will always return same answer for same args.	boolean
Side-effecting	`end-in-self`	A constraint on skipping and order permutation.	EIS group
Abortable	`abortable`	Whether the target is happy to only receive the first word of a transaction.	bool
Abort dynamic energy	`dynamic-one`	Energy used if aborted.	fJ
Complete dynamic energy	`dynamic-full`	Energy used if completed.	fJ
Input hold	`inhold`	Whether inputs need to be held while the unit is busy.	boolean
Output hold	`outhold`	Whether outputs hold their value until next need to change.	boolean
Per-block (component) attributes used for incremental compilation:			
No-re-enter matrix	`excluded-method`	For a method, which others may not be councurrently invoked.	method names
Combinational matrix	`comb-path`	List of from/to pairs of logic delays.	FO4 delays
Per-block (component) attributes used by SoC Render System Integrator:			
Area	`area`	IP Block Silicon Area.	NAND2 equivalents
Static Power	`static-power`	Power consumed when idle.	nW
Frequency	`maxclock`	Maximum Clock Frequency Estimate.	Hz
Portable/Nailed	`nailed`	Whether the block must be on a named die. Otherwise it can be dynamically deployed.	Physical FPGA name
Mirrorable	`mirrorable`	Whether the number of instances is flexible.	arity or 'true'
Off-chip memory cap	`heapspace`	For each logical address space, the amount of store needed.	bytes
Logical Memory Bank	`memgroup`	For each load and store port, name of logical address space.	logical bank id
Bandwidth	... TBD ...	For a port, the expected average data transfer rate.	bits per second
Priority	`priority`	For a port, how important it is that this port has low latency.	dimensionless
Domain name	`domain`	Name of an equivalence class within which data conservation must be maintained.	string
Allocation granularity	`granularity`	Round up to be applied when summing disjoint shared uses of a resource (memoryspace normally).	bytes

TABLE I: The IP-XACT Extensions Used in the current version of the tool.

defined rules based on concepts of data conservation within a domain of interconnection.

The automatic generation axioms are:

- The number of primary IP blocks (one normally), the number of FPGA chips and the number of static external ports are all statically set in the blade manifest (initial configuration). They are all given instance names. Their plurality may not not be adjusted by System Integrator.
- The plurality of all other components may be freely adjusted by System Integrator, but it may not replicate state-bearing components (unless they have mirror rules defined in the future).
- The IP-XACT max-masters and max-slaves attributes are obeyed: ports are either multicast or one-to-one. A target port on a service component may be needed or may be left disconnected. But all initiating ports must be connected to a matching target port.
- The resulting design should give a low value for an overall goal function. This will tend to minimise the number of additionally instantiated components and typically causes them to be wired in tree-like structures to minimise latency.

The general flow for the tool is illustrated in Fig. 4. Its inputs are the name of a primary IP block for the top-level, a search path for lookup of the so-called subsidiary and auxiliary IP blocks, and a description of the target platform described in a file `blade-manifest.xml`. It makes a placement, instantiating auxiliary blocks as needed and then writes out a master RTL file for each FPGA. The whole design is also reported with three further files. These are an IP-XACT report, a graphical plot and a human-readable report that tabulates utilisation metrics and the memory base addresses for each module.

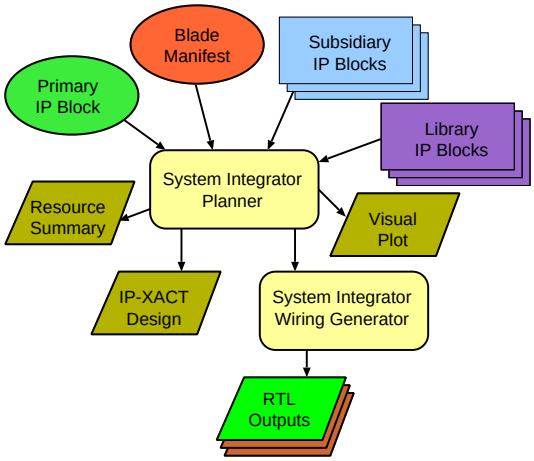

Fig. 4: System Integrator Tool: Inputs and Outputs.

The blade manifest lists the number of FPGAs available on the platform, describing their size, interconnection pattern and hardened IP ports and capabilities. It is an XML file crafted by hand or using an XML editor.

The tool can potentially use any standard optimisation procedure to minimise its global cost metric. The current implementation uses a constructive placer that is run about 50 times using different pseudo-random seeds with the best solution and spread being reported. A critical consideration is whether any IP blocks themselves are good candidates for consequential re-synthesis. There are three reasons for re-synthesising a component:

1) **General time/space fold:** Standard HLS tools have considerable freedom to produce large and fast designs or smaller designs that require a greater number of clock cycles.
2) **Degree of Port Mirroring:** Where a subsidiary block

can be mirrored, the parent needs to be synthesised with a determined number of master ports when these are connected one-to-one with the children. Moreover, the number of load, store and load/store stations on the component can also be manually controlled with our tool.

3) **Move to variable-latency handshakes:** Where a block instantiates a fixed-latency child connection, but then that connection has to be converted to variable-latency owing to inter-FPGA bridges (or perhaps being in a server farm in the future).

The System Integrator's main job is to generate a design that includes the primary IP block and all the support it needs. Starting from the primary IP block, it adds the subsidiary IP blocks referred to in its port list. These may have further application-specific ports (as shown in Figure 5) that in turn need to be supported. Hence it iterates at this stage. Using its constructive placer, it puts each block on a named FPGA where there is sufficient area remaining. Connections that span multiple dies have their necessary protocol adaptors instantiated straightaway. Where a bridge link is shared between bus connections, concentrators are added (addressing tags are later created in a global colouring step). Any placement attempt where any hard limit is breached is aborted without further study. Hard limits include any FPGA being full, as just mentioned, or a guaranteed throughput or latency (sequential or combinational) cannot be met. The placer calls on its pseudo-random number generator each time a non-obvious decision is made, such as which inter-FPGAbridge or memory bank to use for the next wiring step. Possibly a genetic algorithm would neatly deliver good overall designs, but we have not explored that.

As illustrated in Figure 5, there are three forms of bus connection understood by System Integrator:

1) A **Primary Application-Specific Interface** enables a component to invoke functions using a custom bus structure on a child component that has a reverse interface of the same type. In our HLS system, such bus specifications are emitted automatically as augmented IP-XACT bus abstraction documents. The same file is emitted when either side is compiled, with the second simply overwriting the first. When the boundary reflects a class definition in the high-level language, the file name and interface name are the same as the class name. Such a class can have any number of methods and each method will use some set of the busses (or 'ports' as they are called in IP-XACT) making up the interface. This sort of connection is also used for connections to the standard libraries of maths functions.
2) A **Service Interface** provides access to main memory resources for the component. The component is free to instantiate its own RAMs where it wishes, such as FPGA block RAM, but larger regions need wiring to DRAM resources. These are either statically instantiated on the server blade or else accessed over AXI or PCIe on some platforms.
3) A **Directing Interface** provides start/stop control of the primary application and collects status and abnormal end codes from subsidiary blocks. It may also provide debug inspection.

Broadcast connections are only currently supported for and used for static constant values, such as a zone-ID tie off that gives the current layout zone (aka FPGA) number when needed for tagging data. Everything else is a one-to-one connection that is only fanned-in or out by instantiating concentrators and aggregators.

The System Integrator understands the forms of IP block given in Table II.

Every port on any IP block has both an IP-XACT abstraction type (i.e. the name of the protocol) and also a so-called domain name. The abstraction names may be standard bus protocols or auto-generated names for application-specific interfaces that are based on a digest of the interface signature, in the same way that C++ performs linking. The domain names for the application-specific interface between a subsidiary block and its parent is unique and simply ensures a one-to-one connection. But much more freedom exists over the service interfaces. These are generally left blank by the HLS tool (unless the user has inserted as specific pragma), but owing to the structural differences between directorate, memory and other ports, there is no danger of confusion. For each domain, the System Integrator wires everything together, instantiating protocol convertors for inter-FPGA bridging as it goes, thereby achieving conservation of data.

A connection between two components is valid when all of the following conditions hold:

- **Kind Name**: the protocol kinds have the same name. Differences in the other three IP-XACT naming attributes, vendor, version and library name, are warned about but otherwise ignored.
- **Connection Rule**: A one-to-one connection must have two peers: one an initiator and the other a target. A multicast connection must have exactly one initiator.
- **Parameters Match**: IP-XACT parameters are key/value pairs, and these must match apart from any that the user specifically annotates (on the command line) as allowed to mismatch. This ensures, for instance, that a 32-bit data bus is not connected to 64-bit data bus. To overcome simple mismatches of any complexity, one side needs to be manually renamed by the user and an additional protocol adapator added on the search path that encompasses the adaption, such as ignoring unused address bits. Automation of this is expected in the future.
- **Unified Domains**: The connection domains must either already match under the current unification or else a fresh, non-contradictorary, unification is added for the remainder of the design construction.

Many subsidiary IP blocks are mirrorable. The number of instances in use is baked into the HLS compilation of the parent block but the System Integrator can explore the performance benefits of altering the number of instances and report derivatives. It can also try alternative versions of the same component that have been compiled with a different time/space tradeoff. There may be second-order consequences

- Primary IP Block — a top-level component of the design, typically the HLS result from the main application, that embodies an algorithm or processes and generates work for the all the other components.
- Subsidiary IP Block — an IP-block with slave ports that performs an operation. Examples are RAMs, ALUs and HLS outputs from earlier parts of an incremental compilation process. Subsidiary IP blocks may be nailed (statically-instantiated) to a given physical FPGA with fixed instance count, such as a DRAM controller, or may be portable and dynamically deployable on any die.
- Inter-FPGA bridge — statically-instantiated connection between two named FPGAs. This is the one component that is allowed to have its ports in different layout zones.
- Aggregator — for combining ports (typically memories) into a logical entity. These demultiplex based on a chop of an address field.
- Concentrator Pairs — consisting of a tagging muxer and an associated demuxer. The tags are created by our tool and hardwired onto input ports during instantiation and demultiplexing is based on the tag values. Where the available word width is sufficient to carry the data and the tag, these are purely combinational for most bus standards.
- Protocol Adaptor — for converting between bus standards.
- Zone id generator — provides a single, static, broadcast, always-ready, output port number containing the current chip or layout zone number.

TABLE II: Types of Component (IP block) Understood by System Integrator.

of altering the number of mirrors in the parent, but there is no obvious reason why there should be.

Once a design has placed all of the subsidiary IP blocks, it moves on to the second class of our three classes of port connection which is the service interfaces. The main (currently only) service interface used serves random-access storage. Our tool assumes all storage is interchangeable and hence spatial locality is the principle aim of placement. (In practice, some DRAM banks might have a different or non-existent cache structure that we need to countenance in future.)

The algorithm for the service ports: For each domain name, while there is an unconnected initiator, create a connection for it to a suitable serving resource. If the serving resource is an external port that is currently disconnected, a direct connection can be made. But if the external port is already bound, an additional concentrator or aggregator will be instantiated or the arity of an existing one will be increased.

Domain names for physical memory banks may be provided in the blade manifest or else they are dynamically labelled by the System Integrator with unique names. The pre-named approach allows the user to force associations by inserting the same names in pragmas in his high-level code. The System Integrator tool allocates aggregators and concentrators to ensure all service ports are served somehow. It generates a memory map for each resource as it goes (§IV-B).

Once all the service ports are connected, the tool moves on to the directing and debug ports. Their logic is typically minimal but the number of nets needed may vary between 5 and 100 depending on the services provided, varying from a simple abend syndrome for run-time errors, through to some output logging facilities, using virtual LEDs, or full debug access to programmer view registers. The file system access is also via the director at the moment, but this may be changed to be a network-on-chip or other service port. The principle reason for making the director the lowest priority in our constructive placement is that it tends to be the lowest bandwidth and also amenable to being neglected if area issues are pressing. We have not implemented an automatic neglect facility so far.

There are a number of units of measure used by System Integrator to guide its operation and for final reporting. Each has an addition rule for combining. Summation is performed over the tree-structure created. The overall system cost metric, that is the optimisation goal, is a weighted sum of the units of

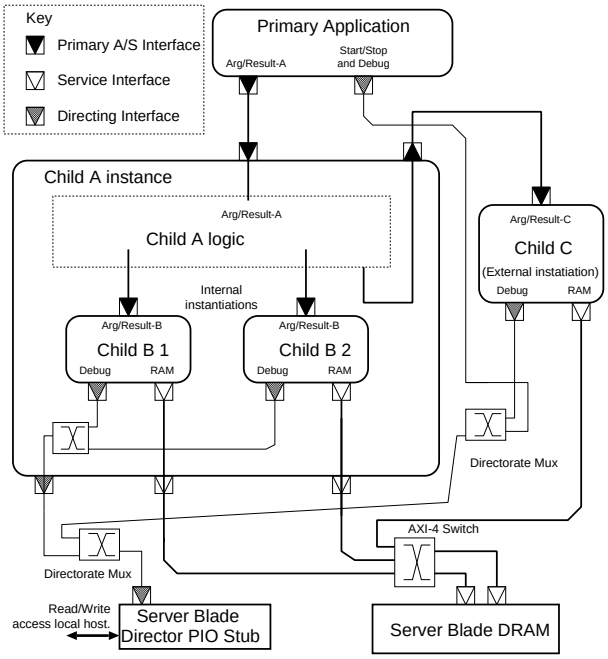

Fig. 5: Characteristic interconnection pattern showing instantiation of subsidiary blocks to serve a primary interface and service inter-wriring for debug and memory access.

measure at the primary IP block. Some weights are normally zero (e.g. the amount of DRAM space used) and others can be fine-tuned by the user on the command line. The units of measure are as abstract as possible in the internal implementation which means that others can easily be added. They fall into classes according to their summation rule. Static power is one of the few that is a simple real number with straightforward addition. The measure units in use are:

- **Static Power** — power in nanowatts when idle.
- **Area** — area in NAND2 equivalents for the system. This is a vector with separate totals maintained for each floorplanning zone, where a floorplanning zone is currently just a physical FPGA.
- **Memory Size** — size in bytes of a memory component or assembly of memory banks when aggregated by auxiliary IP block. Or needed memory size on the backside of a concentrator. The concentrator addition

function has two forms: shared and disjoint. Where ports are providing access to a shared memory there is no increase in size, but where disjoint memories are being stored in one resource there is the normal linear sum, after rounding up to allocation granularity.

- **Average Throughput** — expected or supportable load through a channel.
- **Guaranteed Throughput** — throughput guaranteed to be available: this is required for hard real-time interfaces that must not over-run or under-run.
- **Sequential Delay** — number of clock cycles typical between request and response. Since the latency of inter-FPGA links is several clock cycles, this metric is the most sensitive indicator of whether a mapping of subsidiary IP blocks to physical FPGAs is a good one.
- **Combinational Delay** — our markup on auxiliary block ports enables designs that happen to combine a lot of combinational logic to be poorly scored. The combinational delay addition function is non-linear. It follows the pattern of a static timing analyser that sums the delays of items placed in tandem but applies the maximum operator at a point of confluence.
- **Dynamic Energy** — not used at the moment.

As reported in the lower section of Table I, every block is accompanied with non-functional meta-info that gives an area, latency, throughput and energy cost using IP-XACT extensions.

The system synthesis is guided by a goal function, which is a scalar metric that factors area, delay and energy according to a weights that the user can adjust as desired.

All interfaces are amenable to being bridged and so can be routed over inter-FPGA links, but latency figures are clearly extended and those that were reported as fixed-latency generally convert to variable-latency. A change in latency may affect the choice of best schedule for the invoking parent, meaning it should be recompiled. A change from fixed to variable latency certainly means the parent must be recompiled, since additional holding registers may need to be instantiated for it and other resources.

The service and directing interfaces are amenable to concentrating (aka multiplexing) as well. In general, they could be connected to a network-on-chip, as in the LEAP operating system [4].

Also shown in Figure 5, is that a component instance can be internal or external. External instantiation is where the instance is inside the current (instantiating) module, in the style of a traditional hierarchic design. An external instance is instead formed outside the current module, resulting in additional bindings in the signature of the current module. Although an external instantiation is more verbose, its principle advantage is where the instance has a one or more of service ports that would instead need to be conveyed through the current instance signature.

Other minor tasks that the System Integrator supports are:

1) Allocation of AXI tag numbers which are fed into concentrator component ports as RTL tie-offs to literal constants.
2) Tie-off of unused slave ports on concentrators that have a greater arity than is needed.
3) Memory base address allocation (§IV-B).
4) Writing an IP-XACT design document for the completed system.
5) Writing a SystemC version of the completed system.

A. Blackbox Components

The ability to import separately-compiled components also forms the basis of a **black box** import mechanism for third-party IP blocks. Instantiating a black box containing third-party IP is no different from instantiating a separately-synthesised module. The clock cycles used to operate this component will be fully parallelised alongside other operations concurrently scheduled as part of high-level synthesis.

Third-party IP blocks and existing hardware interfaces are typically described in terms of net-level timing waveforms or formal specifications thereof. To exploit these components from a high-level language via HLS, wrappers need to be manually written and added to the local library. Example third-party black-box components are inter-FPGA links and network ports, the AXI connections to the rest of the SoC on the Zynq platform (`www.xilinx.com`), the CAMs on the NetFPGA boards [5] and the new LUTs as FIFO mode in some FPGA families.

The following example shows the typical structure of such a wrapper. An alternative implementation containing a high-level behavioural model of the resource is also typically needed so the high-level program can run on its own outside the synthesis framework. They both would have the same external signature. The example here shows one direction of the Xilinx LocalLink protocol which is the same in essence as AXI-streaming. Data is transferred on each clock edge where strobe and ready are both asserted (low). The wrapper is compiled once and its resulting IP-XACT file is placed on a folder on the IP block search path. The IP-XACT file contains the filename for the RTL. The RTL is generally very small and can be largely combinational, in which case it is elided with the initiating logic during logic synthesis.

```
class blackbox_wrapper_tx_demo
{
  [OutputWordPort("wdata")] static byte wdata;
  [OutputWordPort("n_wstrobe")] static bool n_wstrobe;
  [InputWordPort("n_rdy")] static bool n_rdy;
  [OutputWordPort("n_sop")] static bool n_sop;
  [OutputWordPort("n_eop")] static bool n_eop;

  [Remote("protocol=HFAST")]
  public static void SendPacket(byte [] darray, int len)
  {
    PauseControlSet(PauseControl.hardPauseEnable);
    for (int i=0; i<len; i++)
      {
        n_wstrobe = !true;
        n_sop = !(i==0);
        n_eop = !(i==len-1);
        wdata = darray[i];
        while (!n_rdy) Pause();
        Pause();
      }
    n_wstrobe = !false;
  }
```

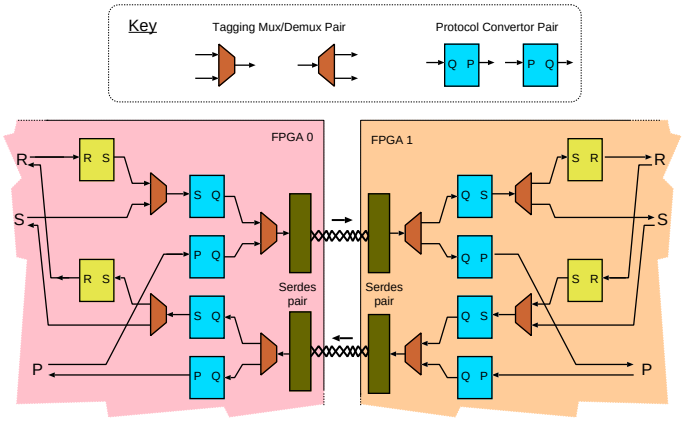

Fig. 6: Inter-FPGA bridge structure: typical setup. The SERDES instances, as described manually in the blade manifest, are utilised by the System Integrator's instantiation of protocol adaptors and concentrators as required.

```
}
```

In this example, markup around the static variables contains instructions to make net-level connections. Actual device pin numbers can be included in the attributes. With this HLS tool, the initial set of pause mode to 'hard' means that logic will be scheduled to consume a clock edge precisely where control flow encounters on of the `Pause` calls. Such adaptors can also be manually written in RTL if preferred.

As illustrated by the SERDES pair in Figure 6, inter-FPGA bridges are bi-directional and have four ports for binding by the System Integrator as it creates an inter-FPGA network. The two ends of each simplex channel have the same domain name, but the bandwidth and latency for the two channels can be described differently in the associated IP-XACT description. Each of the four bus interfaces is AXI streaming with a specified word width, giving the lossless FIFO paradigm. Each direction of the pair is kept matched by the System Integrator, as it adapts the hardware resource to its needs. The adaption steps are just the same as may be freely used elsewhere in the assembly: they are inserting a protocol adaptor pair on each side or inserting a concentrator pair consisting of a tagging mux and an inverse de-multiplexing component that processes and removes the tags. There is a set of standard protocol adaptors corresponding to all basic method signatures of up to 3 arguments with and without a result in our standard distribution. Others can be created by hand as needed and added to the library, or they can be macro-generated on demand in the future. Glue logic for these purposes can also be synthesised from a non-deadlocking, data-conserving product of protocol state machines by known techniques, such as [6].

B. Memory Map Management (Link Editing)

A shared memory resource that is serving a plurality of disjoint requirements needs memory management to statically or dynamically allocate disjoint memory to each component. This is essentially a link editing problem.

The HLS tool will instantiate small local memories using FPGA BRAM. Larger memories must be allocated to off-chip banks. On the Amazon F1-16 platform, 8 FPGAs have a total of 32 separate DRAM banks, each of size 16 GB. Any number of these can be aggregated using programmable logic into a single addressable space with the remainder being kept separate. Hence the number of logical banks and physical banks may differ. In our approach, as stated earlier, each memory allocated by the high-level application code can optionally be marked up with a memory domain (aka bank) name. Where these match named banks in the blade manifest, the System Integrator will allocate memory space of appropriate size in the named bank. The amount of memory needed in each bank is reported in the IP-XACT file for the block using our `heapspace` extension. Where the user has not given names, the System Integrator uses an intelligent algorithm to aggregate sufficient un-mentioned banks to form a logical memory as needed and then serve the memory needs of blocks from their nearest such logical bank.

The base address selected by System Integrator is inserted into the programmable logic via the RTL parameter override mechanism (denoted with hash signs in Verilog) ready for when the die is compiled by the FPGA logic synthesiser.

On the Zynq platform, there is a single memory controller that serves both the ARM cores and the programmable logic. In the simplest approach to using this system, the operating system must be configured to not use memory above a certain address and that address becomes the base address for regions allocated by System Integrator.

An alternative to compile-time allocation is for the operating system to invoke a kernel `malloc()` of physical DRAM and for the application to pass that base address, by programmed I/O over the directing interface, into a register inside the (or each) hardware memory server. The hardware memory server is a pre-defined IP block used in our HLS system for dynamic heap allocation.

V. Prior and Further Work .

The basic tradeoff between partitioning before and after HLS were explored in [7]. They conclude that partitioning before high-level synthesis is the better approach, which is our approach. There is a vast literature covering the partitioning of software over heterogeneous and homogeneous multi-core computing platforms (see for example [8]). Static analysis generally needs to be strongly augmented with profile-directed feedback to select a good partitioning and memory system design. Currently we are using manual annotations of expected memory traffic at each port, but we will automate this in the near future based on extrapolation from small test runs.

A number of extensions to IP-XACT have been proposed in the past. For instance, energy and area extensions are proposed by [9]. In turn, those authors cite other extensions for automatic device driver generation and so on. We do not know of any work where IP-XACT has been extended for automatic HLS link editing as we here do. But kind peers and reviewers, please let us know.

Our contribution is two proposed sets of extensions to the IP-XACT standard for community adoption. One set is needed for IP-XACT-based incremental HLS compilation and the other for completely automatic system assembly and partition over

multiple cards. Both sets are implemented in our prototype tool chain that is available for open source download.

So far, we have not automated as many aspects of the design flow as is potentially possible. In particular, where an IP block needs to be recompiled as a result of the System Integrator's actions or decisions, or because the system has come out too big to fit on the available substrate. It is up to the user to add the flag settings or additional pragma attributes to the compilation stage of that IP block and to manually recompile it. This must be iterated until the whole system builds.

Also, we have not, so far, explored more sophisticated algorithms and metrics for guiding the IP block placement and structure of the interconnect. For instance, when bandwidth use figures and operation activation counts have been measured or estimated, these can be fed into the planning stage. But no substantial change to the design flow or current implementation should be needed.

Creating some forms of network-on-chip is an emergent behaviour from System Integrator, but it will not currently create a ring network. Rings are commonly used in practice, providing full connectivity with minimum of logic. To create a formal ring network an additional component type for the ring 'station' should be added to those understood and suitable implementations added to the IP library. The System Integrator would then route the ring(s) over the inter-chip bridges with a sensible toplogy. The services needed from the network would be extracted as before, using custom adpators, also automatically selected from the the IP library, based on matching bus abstraction type.

Current status: a demo of the integrator tool was presented at FPL-2017. IP-XACT input and output phases that implement our extensions are implemented in the HPR L/S library and accessed by various tools implemented in that library, such as Kiwi, Toy Bluespec, HPR Sytem Integrator and Joiner. 2023 note: incremental HLS experiments are now being conducted in the BlueParrot compiler. See parent web page for links to these projects.

REFERENCES

[1] M. Motamedi, P. Gysel, V. Akella, and S. Ghiasi, "Design space exploration of fpga-based deep convolutional neural networks," in *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*, Jan 2016, pp. 575–580.

[2] "IEEE/IEC international standard - IP-XACT, standard structure for packaging, integrating, and reusing ip within tool flows," *IEC 62014-4 IEEE Std 1685-2009*, pp. 1–373, March 2015.

[3] N. Ramanathan, S. T. Fleming, J. Wickerson, and G. A. Constantinides, "Hardware synthesis of weakly consistent c concurrency," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '17. New York, NY, USA: ACM, 2017, pp. 169–178. [Online]. Available: http://doi.acm.org/10.1145/3020078.3021733

[4] A. M, F. K, Y. Hj, and E. J, "The LEAP FPGA operating system," in *24th International Conference on Field Programmable Gate Arrays and Applications*, Sept 2014.

[5] N. Zilberman, Y. Audzevich, G. A. Covington, and A. W. Moore, "Netfpga sume: Toward 100 gbps as research commodity," *IEEE Micro*, vol. 34, no. 5, pp. 32–41, Sept 2014.

[6] D. J. Greaves and M. J. Nam, "Synthesis of glue logic, transactors, multiplexors and serialisors from protocol specifications," in *2010 Forum on Specification Design Languages (FDL 2010)*, Sept 2010, pp. 1–7.

[7] V. Srinivasan, S. Govindarajan, and R. Vemuri, "Fine-grained and coarse-grained behavioral partitioning with effective utilization of memory and design space exploration for multi-FPGA architectures," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 9, no. 1, pp. 140–158, Feb 2001.

[8] S. A. Ostadzadeh, R. J. Meeuws, K. Sigdel, and K. Bertels, "A multipurpose clustering algorithm for task partitioning in multicore reconfigurable systems," in *2009 International Conference on Complex, Intelligent and Software Intensive Systems*, March 2009, pp. 663–668.

[9] S. Vinco, M. Lora, E. Macii, and M. Poncino, "IP-XACT for smart systems design: extensions for the integration of functional and extra-functional models," in *2016 Forum on Specification and Design Languages (FDL)*, Sept 2016, pp. 1–8.