IPEC: Open-Source Design Automation for Inter-Processing Element Communication

David Volz[1 [0000−0002−1031−2695]], Christoph Spang[1 [0000−0003−1606−4474]], and Andreas Koch[1 [0000−0002−1164−3082]]

Embedded Systems and Applications, Technical University of Darmstadt, Karolinenplatz 5, 64289 Darmstadt, Germany, volz@esa.tu-darmstadt.de

Abstract. With growing FPGA capacities, the complexity of realizable systems-on-chip grows as well. State-of-the-art FPGA accelerators encompass many heterogeneous processing elements that often require efficient Inter-PE communication, as well as with external interfaces, e.g., to the host or memory. While the toolflows and languages to create *individual* processing elements have improved considerably in recent years, the composition of multi-PE SoCs on FPGAs, including the required custom interconnects and the creation of powerful APIs for a host to *interact* with these complex accelerators, has been a largely manual and error-prone ad-hoc process. The IPEC system described here aims to automate much of this effort by offering the system architect selected powerful primitives to easily describe even advanced SoC compositions. Compared to traditional manual approaches, the length of the required descriptions has been reduced by up to two orders of magnitude for the real-world designs examined here. For easy usability, the open-source IPEC system employs a domain-specific language embedded in Python.

Keywords: Automated On-Chip Interconnect · Task Parallelism · Processing Elements · TaPaSCo · FPGA Design Automation.

1 Introduction

Reconfigurable logic devices such as FPGAs have been less affected by Moore's Law slowing down, and continue to offer larger capacities with each new generation. However, apart from specialized applications such as ASIC emulation, actually putting all of that reconfigurable space to good use, e.g., for improved computing performance, remains challenging.

Construction of individual Processing Elements (PEs) improved significantly due to advances in High-Level Synthesis (HLS) and new hardware construction languages, but the assembly of a complete System on Chip (SoC) leveraging many heterogeneous PEs and distributed memory still takes considerable effort.

Some aspects of this complexity have been addressed by abstraction frameworks such as TaPaSCo [9] and others, as discussed in Section 3. These systems can automate much of the lower-level aspects of the SoC construction process,

and provide concise and efficient APIs for interacting with the host, hiding many intricacies of the underlying mechanisms.

What is still lacking is support for more easily describing systems of many *parallel interacting* PEs. While the connections can be created using tools such as Xilinx IP Integrator, this is a laborious process when using the GUI. It is possible to automate that process with Tcl scripts, but manually creating these scripts is similarly tedious and highly error-prone, especially for complex designs.

As an alternative to existing work, we contribute the *Inter Processing Element Communication* (IPEC) framework, for automatically synthesizing complex systems of interacting PEs. IPEC descriptions are formulated at higher abstraction levels than IP Integrator and described in a concise Domain-specific Language (DSL) embedded into Python. The toolflow then leverages the existing TaPaSCo framework to create the lower-levels of the SoC, and also provides automated hardware/software integration.

These high-level descriptions allow an easy scaling of architectures, and thus enhance the portability of the same base-architecture across different device sizes. By creating custom interconnect structures, the area and performance overheads of using a general-purpose Network on Chip (NoC) can be optimized.

Even in its initial form described here, IPEC already enables higher productivity hardware designs by raising the abstraction level and degree of automation over existing solutions. But its underlying technologies, such as the IPEC Intermediate Representation (IIR) used to internally represent entire accelerator-heavy SoCs with their communication and synchronization mechanisms, forms the basis for more advanced SoC-level optimization steps in further development.

Section 2 describes the fundamental ideas, protocols, and components IPEC builds upon. Section 3 gives an overview over related work. Section 4 discusses the primitives provided by IPEC, while Section 5 shows how the user can integrate them into a design. Section 6 demonstrates IPEC for two different use-cases. Section 7 concludes with future work.

2 Fundamentals and Terminology

PEs, in the context of IPEC, describe a computing unit that can be instantiated multiple times. Depending on the individual use-case, a design may either consist of homogeneous or varying purpose heterogeneous PEs. A PE may have access to local and global memories and may be interconnected to other PEs. For use with TaPaSCo, PEs are packaged as IP-XACT blocks [8].

Task Parallel System Composer (TaPaSCo) provides a toolflow to automatically integrate user provided PEs into a *composition*, a set of interconnected PEs, which in the next step can be synthesized onto an FPGA [9]. TaPaSCo uses the notion of a *task* from the heterogeneous computing model, which decomposes large computations into smaller tasks that can run concurrently. Tasks can be started on a PE from the host using the TaPaSCo API, or from other PEs [10].

The **Advanced eXtensible Interface (AXI)** is part of the AMBA (Advanced Microcontroller Bus Architecture) specification and is a freely-available,

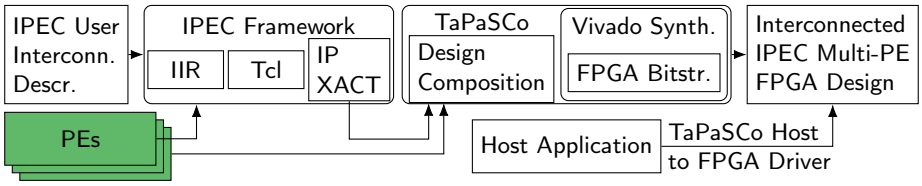


Fig. 1: IPEC Toolflow: User-provided Python-based interconnect descriptions are converted into an IPEC Intermediate Representation (IIR), compiled into Tcl, and packed as IP-XACT core. TaPaSCo gets this core definition and creates the composition, which is synthesized onto FPGA.

open standard for the connection and management of functional blocks in a system-on-chip (SoC) [1]. It is available in older and newer versions and also includes a lightweight and streaming option.

On-Chip Interconnect Topology Existing on-chip Interconnect Topologies are often regular structures such as a ring, star, or fully meshed interconnect. As part of this work, we will focus on user-defined and potentially irregular topologies, with direct connections between PEs.

IPEC's Toolflow is outlined in Figure 1. The user first provides the design's PEs and a high-level Python IPEC interconnect description to the IPEC framework, which then auto-generates the interconnects in Tcl and packages the multi-PE design as a single transparent IP-XACT core. This IPEC core is then supplied to the existing TaPaSCo toolflow, which builds, synthesizes, and loads the bitstream onto an FPGA.

3 Related Work

Different existing tools assist the user in generating FPGA designs comprised from multiple accelerators. As part of this section, we will focus on those tools, which also include active support for inter-PE communication.

TaPa follows a *Task Parallel* approach like TaPaSCo, but focusses more on HLS [3]. It generates PEs by applying HLS to the compute kernels of an OpenCL program. The PEs are placed on the SoC and connected to a *shared ring network*. Each PE can put data onto the network and peek at or pull from other PEs. TaPa provides C++ structs to describe data exchanges between PEs. Using them from within a kernel will result in the corresponding ring network accesses at runtime. Since all communication takes place on the shared ring network, this can pose a *bottleneck* if many PEs are active at the same time and try to communicate.

ESP generates a system from HW tiles [7]. A tile can be a processor, memory, accelerator or auxiliary function. The processor and accelerator tiles include a first level cache and a DMA engine for accessing the caches of other tiles, while memory tiles only include a last level cache. DMAs are routed over a mesh network connecting all tiles. However, a potentially more efficient *direct communication* between accelerators is *not supported*.

GENIE (GENeric Interconnect Engine) enables the user to connect compute elements (CEs) according to user defined connection tables [16]. Hardware for splitting and merging connections is generated automatically. It uses a custom *routed streaming* protocol, which includes addressing by giving each slave an ID. However, it only generates interconnect structures for its own protocol and requires that CEs support it.

Archborn provides a Tcl abstraction layer, which enables the user to concisely create PEs and Memories, then connect them using busses [14]. The user can attach PEs to a bus, which in turn can be connected to create an NoC. However, the user has to manually create these structures and the framework does not include conversion between protocols.

Cascabel is a TaPaSCo extension, which enables on-device dynamic dispatch [10]. It replaces the default TaPaSCo scheduler with one that can process launch requests from host and PEs. However, it can only launch one task at a time, each with a limited number of task parameters. Already running PEs *cannot communicate directly*.

A key limitation of the provided frameworks is that they mostly rely on fixed protocols and interconnect topologies (e.g., mesh, star, ring). While this does make sense for ASICs, the large multiplexers are often slow on FPGAs. In contrast, IPEC generates *custom interconnect structures* matching the communication patterns of a specific application.

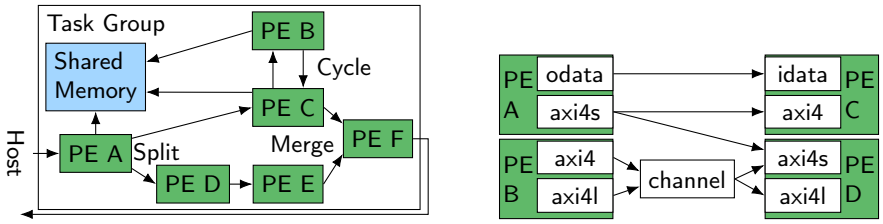
4 Capabilities

With TaPaSCo and other task parallel frameworks, the fundamental abstraction is that of a *task*, which is submitted to one PE and processed in its entirety. IPEC, on the other hand, uses *task groups* as the fundamental abstraction. A *task group* comprises multiple tasks, which can exchange data using shared memory or connections. Figure 2a shows such a task group as a Data-Flow Graph (DFG).

Each of the PE A to F represent one task of the entire group. Each task, in turn, runs on a PE optimized for it. The host launches a task group together with the necessary input parameters and is notified of its completion by using the interrupt signal of a designated PE, usually either the first or last one in the DFG. PEs can share data only along the edges of the DFG, this means that, e.g., PE D has no means of communicating with PE C. Note that the DFG is a directed graph, but which may contain cycles.

4.1 Connections

Every PE has one or more ports to send data to or receive data from other PE's ports and memory. IPEC supports three protocols from the AXI4 family for the ports: AXI4, AXI4 Lite, and AXI4 Stream. Additionally, connecting individual wires is possible as well. Using IPEC, the user can create arbitrary connections between ports, e.g., as shown in Figure 2b.



(a) Possible data-flow graph of a task group: After launching a task group from the host, the PEs (A - F) communicate during computation. Each edge represents an individual connection.

(b) Sample topology between the ports of PEs (A - D): IPEC is flexible regarding a port's fan-in, fan-out, and protocol. Routing resources can be reduced by consolidating connections into channels. Connecting individual signals is also supported.

Fig. 2: A sample data-flow graph and topology, which can be mapped using IPEC.

A single master port can have multiple outgoing connections to other slaves, while a single slave can have multiple incoming connections from other masters. After specifying all connections, IPEC automatically generates the corresponding hardware and connections further described in Section 5.5. Interconnects are created automatically when needed. For ports with many incoming or outgoing connections, the design is automatically optimized by creating a hierarchical interconnect structure. With address-based protocols like AXI4 and AXI4 Lite, a master can send data to individual slaves by using an address map (described in greater detail in Section 5.6). AXI4 Stream masters, on the other hand, broadcast data to all of their slaves. AXI4 Stream slaves include a FIFO in order to act as a buffer to avoid slowing-down the master in case a slave is not ready yet.

Furthermore, connecting ports with mismatching protocols is supported, but may require the user to specify a conversion protocol. Converting AXI4 to AXI4 Lite is done automatically, while converting an AXI4 Stream to AXI4 requires the user to specify a hardware module to do the conversion.

In order to save routing resources, connections between different ports can be grouped to form a *Channel*. A Channel can have multiple input ports, which are arbitrated onto a single connection. On the other end, the data is forwarded to the single addressed slave, or broadcast in the case of streams.

4.2 Memory

One advantage of FPGAs is the availability of distributed memory and customizable memory systems such as [11]. To exploit distributed memory, PEs can each have a local BRAM attached, which can also be made accessible to other PEs. Since BRAM on most modern FPGAs is dual-ported, IPEC exposes both ports to the user, who can then decide whether to give a single PE exclusive access to a port for minimum latency, or share it among multiple PEs. Furthermore, IPEC treats all types of memory identically, including BRAM, DRAM, HBM,

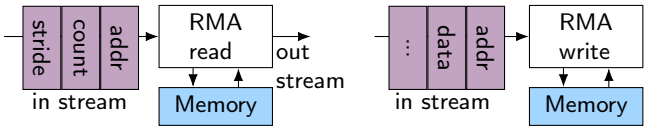


Fig. 3: Shared memory accesses using AXI4 Stream, require RMA converters, which are auto-generated by IPEC. An RMA read has up to three parameters determining which data words to return. An RMA write has at least one data packet, but will write additional data to successive memory addresses.

and register files. In the case of off-chip memory, IPEC delegates generating the memory controller to TaPaSCo, but will create the necessary interconnects for merging all connections accessing the same memory. PEs can perform their memory accesses via AXI4, AXI4 Lite, and AXI4 Stream. For AXI4 Streams, the required Remote Memory Access (RMA) converter units, as shown in Figure 3, are created automatically.

RMA Read The PE has an address stream connected to the RMA unit and a data stream back from the RMA unit. A read request contains up to three parameters: the address, number of elements to read and the stride between successive elements. After receiving a read request, the RMA unit will read the data from memory and broadcast it to all receivers of the stream.

RMA Write The PE has a combined address and data stream connected to the RMA unit. A write request contains at least two parameters: the address and the data. Sending more than one data packet will write to successive addresses.

Data stored inside a PE and shared memory is persistent across launches, meaning it is still available when the next task or task group is started. However, it falls to the user to ensure that, if a later task requires data from a previous one, the new task or group is launched on the specific PEs that can physically access that memory.

4.3 Dispatch - Starting PEs

Since a task group consists of multiple tasks, and therefore involves multiple PEs, there has to be a way to start all PEs belonging to the same task group. We discuss the two possibilities shown in Figure 4.

Software Dispatch Task groups can be launched under host control by assigning each involved PE a unique ID. The host can then individually launch the tasks in a group using the identified PEs. While very flexible, software dispatch has a relatively high communication overhead, as each task requires two PCIe transfers. In the example at the top of Figure 4, PEs A and B are launched as group under host control. Afterwards, their respective tasks can communicate using IPEC facilities.

Hardware Dispatch Instead of being launched under host control, IPEC can configure PEs to be launched on-chip, without host intervention. Therefore, PEs are fitted with a *Stream Starter*, which accepts new launch requests, including

the required parameters, from an AXI4 Stream. This allows arrangements as shown in the left part of Figure 4, where only PE C is host-launchable by its ID. The PEs D is then launched on-chip over IPEC links. In the example, PE D can receive additional data from and return its result to PE C over additional streams. The Stream Starter blocks until its controlled PE becomes idle again. Note that hardware dispatch requires some care from the user, as in some cases, such as circular structures, there is the risk of deadlocks.

Locks for Stream Synchronization To prevent multiple PEs from interfering with each other, including deadlocks, individual stream connections can be blocked using a lock. Locks contain an accumulator register connected to an AXI4 Stream slave. Each incoming packet increments the accumulator or applies a simple binary operation with the data field of the packet. This way, accumulators can realize *atomic operations* across multiple concurrently executing PEs. Additionally, the lock is linked with a channel and blocks all communication over the channel if the accumulator is non-zero. This allows the realization of different synchronization schemes such as semaphores, and mutexes.

5 Using IPEC to Simplify SoC Implementation

This section details how the user can create compositions using the previously discussed functionalities. For ease of use and to make it more accessible to users inexperienced in hardware development, IPEC is controlled by high-level *Python* descriptions. We chose Python specifically for lowering the hardware designer's entry barrier. With IPEC, not only single-PE but multi-PE HLS designs become feasible without the need for Tcl or any HDL knowledge to interface with existing toolflows. Together with TaPaSCo's HLS support, the user can now easily generate a multi-PE SoC with custom interconnect structures.

As shown previously in Figure 1, using Python syntax, the user writes a script instantiating all PEs, memories, and connections between them. Our library then converts the given description into an intermediate representation, from which the necessary converters and interconnects can be automatically inferred and the address map is computed. Finally, from the extended intermediate representation, an IP-XACT core containing all resources is created. For maximum automation, IPEC is integrated into TaPaSCo. However, it can also be used in a stand-alone

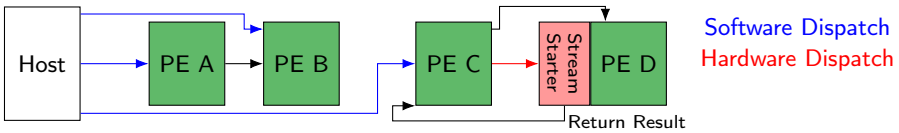


Fig. 4: IPEC supports both host software and on-chip hardware dispatch. With software dispatch, the host starts PEs individually. With hardware dispatch, PEs have a Stream Starter attached to their configuration registers, which allows other PEs to start tasks without host intervention.

Listing 1.1: Python code for an introductory example of an IPEC composition.

```
1  pea = PEA(ID=1)
2  peb = PEB()
3  pec = PEC()
4  hbm = HBM()
5  c = Channel(pea.maxis0,peb.config)
6  lock = Lock(c)
7  Channel(peb.maxis,pec.config)
8  Channel(pec.maxis,peb.config)
9  Channel(pea.maxis1,pec.maxis1,lock)
10 for i in range(0, 10):
11   ped = PED(ID=i+1)
12   bram = BRAM('16K')
13   Channel(ped.maxi0, bram.port0)
14   Channel(ped.maxi1, hbm.saxi)
```

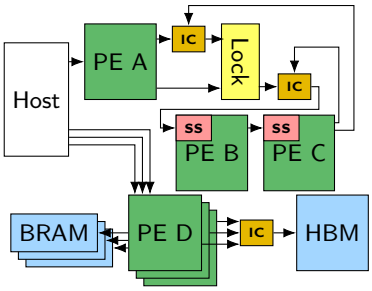


Fig. 5: Corresponding block design: PE B and PE C form a cycle started by PE A through a lock. Each PE D has its own BRAM and a connection to shared HBM.

manner in other design flows. Listing 1.1 is a simple introductory example for using IPEC to describe the block design shown in Figure 5. More complex real-world use-cases will be discussed in Section 6.

5.1 Device, PEs and Memory

When using IPEC, PEs can be instantiated by calling a Python constructor via its identical name. To this end, IPEC reads the user's TaPaSCo hardware cores directory and automatically creates a Python class for every PE type found. Note that IPEC adheres to TaPaSCo's tenet of being language agnostic. Thus, while the actual cores might have been created using Verilog HDL, Chisel, HLS, Bluespec, or any other design flow, this no longer plays a role in their IPEC composition. The code in Listing 1.1 creates the PEs PE A (Line 1), PE B (Line 2), PE C (Line 3), and multiple instances of PE D (Line 11). Each PE object contains member attributes for every interface the PE exposes with the same name and protocol, thus making it easy for the user to reference a specific PE port in the IPEC script. Other hardware modules are created similarly, for example the yellow lock in Figure 5 results from Line 6 in Listing 1.1.

In the case of PE A and PE D, the constructor includes the ID parameter, thus making the PEs host-launchable using the TaPaSCo API. PE B and PE C, on the other hand, are equipped with a PE Stream Starter created implicitly by connecting an AXI4 Stream to master to their configuration registers. They can thus be launched on-chip without host intervention.

Memory instances are created similar to PEs by calling a Python constructor of the same name. *BRAM* creates one block of BRAM of the specified size with two access ports (Line 12). *DRAM* and *HBM* aggregate all connected ports to a single interface, which can later be connected to a memory controller.

5.2 Connections

In Listing 1.1, different ports of PEs and memory are connected by creating a channel. At least one master and one slave is to be specified per channel. If multiple masters are part of a channel (Line 9), IPEC creates an interconnect to arbitrate them onto a single connection. If multiple slaves are present in a channel, the resulting interconnect will *broadcast* to all slaves for the case of the slaves being AXI4 Streams, or unicast to the single specific slave as addressed by AXI4 and AXI4 Lite interfaces. Multiple channels connected to the same port are handled analogously. E.g., all PE D instances are connected to the same HBM in Line 14. When creating the lock in Line 6, IPEC splits the given channel and routes it through the lock.

5.3 Locks - Deadlock Avoidance

Locks offer a way to synchronize the execution of PEs, or prevent a deadlock. In Figure 5, PE A can start a cycle, which contains PE B and PE C, with PE B starting PE C as it finishes, and vice versa. If PE A restarts PE B before this cycle is over, PE C will wait for PE B to finish, which, in turn, waits for PE C to finish, causing a deadlock. To prevent this, we add a lock in Line 6.

The channel in Line 9 connects PE A and PE C to the accumulator part of the lock, allowing them to block the second channel through the lock. This second channel allows PE A to start PE B, if the accumulator register is 0. After PE A starts PE B, it increments the accumulator of the lock by one, thus blocking any of its *own* future attempts of starting PE B. PE C's way to start PE B (Line 8), on the other hand, is not blocked by the lock and it can still restart PE B. When no more iterations between PE B and PE C are required, PE C will reset the lock, thus re-enabling PE A to launch PE B again, and start a new processing cycle.

5.4 IPEC Intermediate Representation

For improved efficiency, the IPEC framework does not immediately generate the corresponding hardware when the Python call is processed. Instead, the calls construct the IPEC Intermediate Representation (IIR, see Figure 1) in the background, which is then processed in its entirety to generate the interconnects. The IIR is graph-based and comprises *cells*, *ports*, and *connections*:

Cells are hardware modules available to IPEC as IP-XACT cores. While all cells have ports to express their connection points, IIR distinguishes between *PEs*, *Memory*, *Stream Operations*, *Interconnects*, and *(AXI) Converters*.

Ports encapsulate all the low-level physical signals of a Cell associated with a specific protocol into one user accessible object. IPEC differentiates between AXI4, AXI4 Lite, AXI4 Stream, Clocks, Resets, Interrupts, and raw input and output signals. Address based protocol ports are part of the address map generation.

Connections are point to point connections between ports. IPEC generates a Tcl script from IIR, which, when used from within the context of the Xilinx IP Integrator, *imports*, *creates*, and *configures* every referenced IP and the *connections* between them to generate an IP-XACT core.

5.5 Interconnect Generation

When the IIR is constructed, the protocols of connected ports may not match. Ports may even have multi-protocol fanouts. This is resolved by first inserting generic interconnects which lack all protocol information. Afterwards, protocol information, such as type, data width, and address width, is *propagated* through the connection graph, starting from each port with a known protocol. When different protocol information is propagated to the same port, IPEC selects a common "super" protocol on the master side, or a common "sub" protocol on the slave side. As an example, in Listing 1.1, the data width of the lock is not specified. If multiple masters propagate different data widths to the same slave having an unknown data width, then the *widest* data width of all masters is selected for the slave. Conversely, if multiple slaves connect to a single master, the narrowest data width is selected. The user can control this process by specifying the protocol manually at key points, and then leave it to IPEC to infer the remaining protocol parameters. After protocol propagation, IPEC inserts *converters* between each pair of ports with mismatching protocol specifications. Finally, each generic cell is replaced by one of the IP-cores that is available to IPEC and has the required properties. While these capabilities are similar to those in Xilinx Vivado, they allow IPEC to parameterize generic user IP cores like the lock as needed.

5.6 Address Map Generation

In contrast to AXI4 Streams, AXI4 and AXI4 Lite require assigned addresses for communication. Thus, each master has an address space containing the address segments of every slave it is connected to. IPEC automatically creates these address maps for each master by inserting the slaves in the order specified by the user into the masters address space. IPEC may leave parts of the address space unassigned to ensure each slaves' address segment starts at an aligned address divisible by its own size to allow more efficient address decoding.

5.7 Advantages of Embedding IPEC in Python

IPEC profits from being a Domain-Specific Language *embedded* into a high-level language, namely Python. This is exploited in Listing 1.1 by using a *for*-loop to concisely create multiple PEs, BRAM, and the connections between them. Note that only a single HBM instance is created (Line 4), which has *multiple* connections created in Line 14. IPEC will create an interconnect block *merging* all of the individual links.

While these abstractions could be implemented in Tcl, which is directly supported by Vivado, we chose Python to accommodate non-hardware designers, who can use it in combination with *HLS* to employ reconfigurable computing.

6 Evaluation

This section discusses two *real-world* case studies which successfully leverage IPEC. Both projects initially relied on the Xilinx *IP Integrator* GUI to manually

create communication structures between PEs, before being migrated to IPEC. For both use-cases, other approaches would either not provide sufficient bandwidth (e.g., a single global shared memory), or require more FPGA resources (e.g., an NoC in soft-logic). The final composition when using IPEC is identical to the previous manually created composition. Since the design, and therefore the performance is identical, the focus of this study will thus be on the *productivity gains* achievable using IPEC, as compared to the Xilinx IP Integrator GUI. As simple measure for the productivity gains, we compare the lines of IPEC Python code with the number drag-and-drop user actions required in the GUI, which is reflected by the number of corresponding Tcl lines automatically created by the IP Integrator tool. In our experience, the number of Tcl lines is a good estimation for the number drag-and-drop operations, ignoring grouping commands.

By using IPEC, it becomes much easier to perform design space exploration by varying parameters of a composition, or to scale an IPEC design up or down to target different FPGAs.

6.1 Case Study I: neoDB Database System

Many modern Data Base Management Systems (DBMSs) use *multi-versioning* to enable consistency and high parallelism for both long-running analytical queries (reads), and low-latency update transactions (writes) [18]. In this scheme, the DBMS holds multiple versions of the same tuple linked with timestamps to determine which single version is the current one (*visible*) to a given transaction or query. This *visibility check* requires loading tuples and comparing their timestamp against the timestamp of the ongoing query.

In practice, the number of active versions can reach several hundred millions [12], resulting in many entries being evicted from fast memory to cold storage. Thus, in today's DBMSs, analytical queries may be slowed by high latency memory accesses when checking tuples for visibility.

neoDB is a next-gen DBMS based on PostgreSQL that uses FPGA-accelerated Near-Data Processing (NDP) to address many traditional bottlenecks [2]. The example employed as use-case for IPEC performs visibility checks on the FPGA in NDP-fashion to determine the visible records. These can then be returned back to the host, or be forwarded to further NDP accelerators on the FPGA. In both cases, the results are written back to host memory.

The neoDB composition examined here comprises four types of PEs working together to perform an NDP-operation. MicroBlaze softcores load tuples from memory and perform the visibility checking, forwarding only the visible tuples over an AXI4 Stream. Next, a specialized NDP accelerator continues to process the visible tuples, using a complex data analytics operation in the actual system. The third PE transforms the analytics accelerator results in preparation for writing them to host memory, which is performed by the final PE. These four PE types form a *cluster*, which can be replicated on larger target FPGAs. The host only interacts with the first PE of each cluster, launching parallel tasks on multiple softcores.

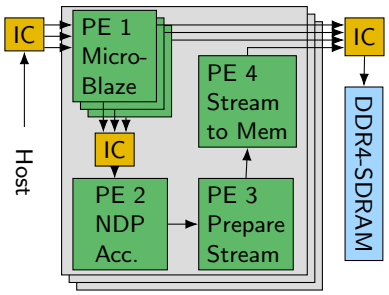


Fig. 6: System composition of neoDB. Configuration connections are not shown for clarity.

Listing 1.2: IPEC code for NeoDB

```
1  dram   = DRAM()
2  for i in range(0,2): # 0..1
3   pe2 = NDP_Accelerator(ID=2)
4   for j in range(0,16): # 0..15
5    pe1 = MicroBlaze(ID=1)
6    Channel(pe1.maxis, pe2.saxis)
7    Channel(pe1.maxi, dram.saxi)
8   pe3 = Stream_Preparation(ID=3)
9   pe4 = Stream_to_Memory(ID=4)
10  Channel(pe2.maxis, pe3.saxis)
11  Channel(pe3.maxis, pe4.saxis)
12  Channel(pe4.maxi, dram.saxi)
```

Figure 6 shows an example of a composition for this architecture, including multiple clusters each containing multiple softcores and an analytics accelerator each. Listing 1.2 shows the IPEC code describing this composition, having two clusters and 16 MicroBlaze softcores per cluster. The Tcl script to generate the composition contains almost 1,000 lines, each line representing one manual and error prone action the user performed when using the Xilinx IP Integrator. Even when using TaPaSCo to create much of the low-level infrastructure, more than 200 Tcl lines remain just to realize the inter-PE communication patterns. IPEC can express these in just 12 code lines and allows to flexibly balance the different processing pipeline parts to match throughputs across stages.

Furthermore, in the future, neoDB will require far more complex communication topologies, as well as support for fast atomic operations for synchronization. IPEC already supports both of these functionalities.

6.2 Case Study II: Hardware Fuzzing Accelerator

Fuzzing is an automated method for finding vulnerabilities in applications using a large number of computer generated test cases as input [15]. A black-box fuzzer will randomly create such test cases and then externally observe the program for unexpected behavior. In contrast, gray-box and white-box fuzzers indirectly or directly obtain the program-internal state. Control flow information helps guiding test case generation towards higher coverage, thereby increasing the chance to actually find unintended program states or vulnerabilities [13].

Traditionally, software-based fuzzing frameworks such as AFL++ [6,5] are used to perform this fuzzing-process. A program is iteratively executed and monitored for any still undiscovered and possibly hazardous state. In software, the monitoring aspect is typically realized by statically or dynamically inserting new tracing instructions into the program. When fuzzing a program in a non-native Instruction Set Architecture (ISA), an emulator has to be employed.

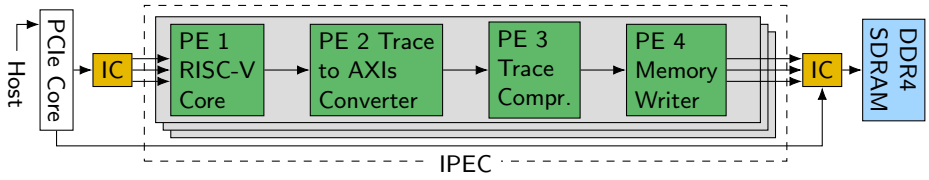


Fig. 7: System Diagram of the Fuzzing Accelerator. Note that a fuzzing cluster of PE 1-4 can be replicated on larger chips for higher fuzzing throughput. Since PE 2-4 are only configured and started by the host once, their connections to the host interconnect are omitted for clarity.

Alternatively, the process of patching, emulating program execution, and monitoring program states can be accelerated on an FPGA. In this manner, a potentially large number of ISA-native processor cores may execute the program with higher efficiency than emulation could achieve. Also, the program state can be monitored via dedicated hardware blocks without the need for special instructions. Beyond uncompressed monitoring, dedicated hardware may also generate and continuously update a condensed trace of the program execution and finally write it into the FPGA's DDR-SDRAM. This coverage information then guides the host in its generation of new and tighter test cases.

In its current version, the fuzzer is limited to executing baremetal-only applications without the capability of including any non-statically linked libraries. Even with these limitations, though, it is suitable to demonstrate IPEC's capabilities.

The fuzzing accelerator is organized into clusters, each holding four communicating PEs as shown in Figure 7. The first PE contains the processor core and a tracing interface. Depending on the specific core's capabilities, the Real-Time Lightweight Integrity enForcement intErface (RT-LIFE) [17] or the RISC-V Trace (interface) Specification [4] are used to monitor the instruction stream. The raw trace output is hardwired to the second PE, which transforms the trace into an AXI4 Stream. The trace is compressed in the third PE. Finally, the fourth PE writes the compressed tracing information into the FPGA's DDR-SDRAM. Each PE communicates with the next one via a hardwired connection, or an AXI4 Stream, while the last PE is connected to DRAM.

The block design for a system containing 25 fuzzing clusters requires a total of 2,000 lines of Tcl to describe, each representing one user interaction with the GUI. Even with the automation already provided by TaPaSCo, up to 600 *additional* design elements have to be manually formulated just for the inter-PE communication, requiring a line of Tcl for each element. The IPEC automation reduces this description to just 10 lines, shown in Listing 1.3. The resulting block design with 25 fuzzing clusters is shown in Figure 8, highlighting the error-proneness of the manual process.

Scaling the number of fuzzing clusters up or down becomes trivial when using IPEC. Combined with the existing TaPaSCo framework, this enables a high degree of portability and very simple design space exploration.

Listing 1.3: IPEC code for 25 fuzzing clusters.

```
dram   = DRAM()
for i in range(0,25): # 0..24
 pe1 = RISC_V_Core_PE(ID=i*4+1)
 pe2 = Converter_PE(ID=i*4+2)
 pe3 = Compressor_PE(ID=i*4+3)
 pe4 = MemoryWr_PE(ID=i*4+4)
 Channel(pe1.o_data, pe2.i_data)
 Channel(pe2.maxis, pe3.saxis)
 Channel(pe3.maxis, pe4.saxis)
 Channel(pe4.maxi, dram.saxi)
```

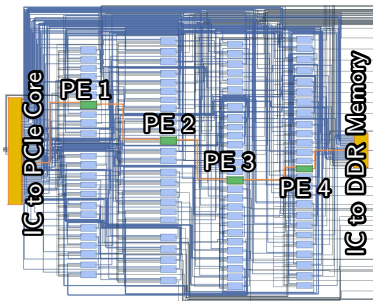


Fig. 8: Resulting block design.

As a result, IPEC allowed to explore compositions with a varying number of fuzzing clusters, and to select the composition yielding the highest wallclock throughput, i.e. fuzzing jobs-per-time, by trading-off parallelism and achievable clock frequency for different FPGAs.

7 Conclusion and Future Work

IPEC provides a solution for building complex SoCs with many interconnected accelerator units. For the two real-world use-cases discussed here, the tool already has significantly improved designer productivity and will enable much more comprehensive design space explorations than feasible using the traditional manual approaches.

Future work on IPEC will build on the existing foundations. Specifically, we will examine high performance off/on-chip task dispatch using the hardware structures introduced in this work, and extending its capabilities to start a predefined set of multiple PEs with a single launch command, including some form of on-device scheduling to maximize utilization of available PEs.

IPEC will be released as open-source software under the GNU LGPL v3 license at https://git.esa.informatik.tu-darmstadt.de/ipec/ipec.

Acknowledgements The authors acknowledge the financial support by the Federal Ministry of Education and Research of Germany in the project "Open6GHub" (grant number: 16KISK014).

Part of this research work has been funded by the German Federal Ministry of Education and Research and the Hessian Ministry of Higher Education, Research, Science and the Arts within their joint support of the National Research Center for Applied Cybersecurity ATHENE.

References

1. ARM: Amba specifications (2022), https://developer.arm.com/architectures/system-architectures/amba/specifications, [Online; accessed 4-January-2022]

2. Bernhardt, A., Tamimi, S., Stock, F., Heinz, C., Tobias Vinçon, C.K., Koch, A., Petrov, I.: neodb: In-situ snapshots for multi-version dbms on native computational storage. Proc. ICDE (2022)
3. Chi, Y., Guo, L., Lau, J., Choi, Y.k., Wang, J., Cong, J.: Extending High-Level Synthesis for Task-Parallel Programs. In: 2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM). pp. 204–213 (2021). https://doi.org/10.1109/FCCM51124.2021.00032
4. Div.: RISC-V trace specification (2021), https://github.com/riscv/riscv-trace-spec
5. Div.: Afl++ github repository (2022), https://github.com/AFLplusplus/AFLplusplus, [Online; accessed 5-January-2022]
6. Fioraldi, A., Maier, D., Eißfeldt, H., Heuse, M.: AFL++: Combining incremental steps of fuzzing research. In: 14th USENIX Workshop on Offensive Technologies (WOOT 20). USENIX Association (Aug 2020)
7. Giri, D., Chiu, K.L., Eichler, G., Mantovani, P., Carloni, L.P.: Accelerator Integration for Open-Source SoC Design. IEEE Micro **41**(4), 8–14 (2021). https://doi.org/10.1109/MM.2021.3073893
8. Group, S.S.W.: IEEE 1685-2009 - IEEE Standard for IP-XACT, Standard Structure for Packaging, Integrating, and Reusing IP within Tool Flows (2022), https://standards.ieee.org/standard/1685-2009.html, [Online; accessed 10-January-2022]
9. Heinz, C., Hofmann, J., Korinth, J., Sommer, L., Weber, L., Koch, A.: The TaPaSCo Open-Source Toolflow. Journal of Signal Processing Systems **93**, 545–563 (2021). https://doi.org/https://doi.org/10.1007/s11265-021-01640-8
10. Heinz, C., Koch, A.: Supporting on-chip dynamic parallelism for task-based hardware accelerators. In: Applied Reconfigurable Computing. Architectures, Tools, and Applications (ARC) (2021)
11. Lange, H., Wink, T., Koch, A.: Marc ii: A parametrized speculative multi-ported memory subsystem for reconfigurable computers. In: ACM Proc. Design, Automation, and Test in Europe (DATE). ACM (2011)
12. Lee, J., Shin, H., Park, C., Ko, S., Jaeyun, N., Chuh, Y., Stephan, W., Han, W.S.: Hybrid garbage collection for multi-version concurrency control in sap hana. pp. 1307–1318 (06 2016). https://doi.org/10.1145/2882903.2903734
13. Liang, H., Pei, X., Jia, X., Shen, W., Zhang, J.: Fuzzing: State of the art. IEEE Transactions on Reliability **67**(3), 1199–1218 (2018). https://doi.org/10.1109/TR.2018.2834476
14. Ma, S., Ding, H., Huang, M., Andrews, D.: Archborn: an open source tool for automated generation of chip heterogeneous multiprocessor architectures. In: 2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig). pp. 1–6 (2015). https://doi.org/10.1109/ReConFig.2015.7393293
15. Oehlert, P.: Violating assumptions with fuzzing. IEEE Security Privacy **3**(2), 58–62 (2005). https://doi.org/10.1109/MSP.2005.55
16. Rodionov, A., Biancolin, D., Rose, J.: Fine-grained interconnect synthesis. ACM Trans. Reconfigurable Technol. Syst. **9**(4) (aug 2016). https://doi.org/10.1145/2892641, https://doi.org/10.1145/2892641
17. Spang, C., Meisel, F., Koch, A.: RT-LIFE: Portable RISC-V Interface for Real-Time Lightweight Security Enforcement. In: Intl. Conf. on Embedded Computer Systems: Architectures, MOdeling and Simulation (SAMOS). Springer International Publishing (2021)
18. Özcan, F., Tian, Y., Tözün, P.: Hybrid transactional/analytical processing: A survey. pp. 1771–1775 (05 2017). https://doi.org/10.1145/3035918.3054784