# PushPush: Seamless Integration of Hardware and Software Objects Via Function Calls over AXI

Shane T. Fleming*, Ivan Beretta*, David B. Thomas*, George A. Constantinides*, and Dan R. Ghica†
* Dept. of Electrical and Electronic Engineering, Imperial College London, UK,
{shane.fleming06, i.beretta, d.thomas1, g.constantinides}@imperial.ac.uk
† School of Computer Science, University of Birmingham, UK, d.r.ghica@cs.bham.ac.uk

***Abstract*—FPGA systems are moving towards a system-on-chip model, both at the architectural level and in the development tools. Developers are able to design and implement IP using a mixture of HLS, RTL, and software, then integrate them with third-party IP cores and hardened CPUs using one or more shared memory buses. This allows functionality to be easily connected together at the bus level, but accessing IP core functionality requires designers to support each component's protocol and co-ordinate hardware from a CPU. This paper presents a protocol called PushPush, which allows HLS, RTL, and software components to expose functionality as strongly typed functions, and allows any component to access functions exposed by any other component in the system. The protocol is designed for maximum efficiency in memory buses such as AXI and Avalon, reducing each function call to two burst writes delivering both data and control, minimising bus traffic and eliminating the need for global polling or interrupt delivery. We demonstrate this approach in a Zynq environment, using components written in C++ (ARM/Linux), C (Microblaze), Vivado HLS (Logic), and Verity (Logic). We show that any component can call functions exposed by any other component, without knowing where or how that function is located. Performance is at least 1 million function calls/sec between any pair of components, and rises to 4 million function calls/sec between pairs of Vivado HLS components.**

## I. Introduction

The steady increase in resources per FPGA allows increasingly complex systems to be developed, but also requires developers to find new ways of overcoming the design-gap. Some techniques have been borrowed from ASIC approaches to the problem, such as the development, sale, and integration of increasingly complex IP blocks. Other techniques such as High-Level Synthesis (HLS) tools are arguably driven by FPGA research, though they are increasingly used in both FPGAs and ASICs. The general trend has been towards productivity over performance, using HLS and off-the shelf IP where possible, and only dropping to low-level tools and languages in performance critical parts.

This trend is reflected in changing system architectures, both at the hardware architecture and the design and IP levels. Both are moving towards a System-on-Chip approach, with hardened processors connected to accelerators in hardware and off-chip memory via one or more shared buses, such as AXI or Avalon. At the bus level the integration works well, making it easy to connect blocks via point-and-click or scripted interface, with automation handling conversion between bus-standards, and inserting or optimising crossbars.

This research was supported by EPSRC grant EP/K015214/1

Beyond the bus level the functional integration approach can be split into tightly integrated automated approaches, such as Altera's OpenCL [1] and LegUp [2]; or ad-hoc loosely-coupled manual approaches such as LEAP [3] and RIFFA [4], where the designer is responsible for developing the protocol glue for connecting hardware and software. The automated approach enforces a software-as-control, hardware-as-accelerator approach, limiting the types of system that can be built: why can hardware not be used to schedule software accelerators? The manual approach offers great flexibility, but removes many of the benefits of bus-based design and HLS, as integration and testing is complex. Both approaches also have significant overhead for each call to an "accelerator", requiring work units to be relatively large to amortize call overhead.

This paper presents a protocol for supporting high-level functions calls between cores within an SoC, whether they are software or hardware. While the main principles of the protocol were outlined in our previous work [5], we herein focus on its implementation aspects, whose key ideas are:

- Addresses as function identifiers: anyone with the bus endpoint (address) of a function can call it directly.
- Global and Local Endpoints: globally writeable addresses are used to pass data to call endpoints, while local read-write addresses are used by the receiver to detect calls.
- Bursts as function calls: both data and control are transferred in one burst to the target function's endpoint, and a single burst can be used to signal function completion.

The system has been implemented in a Zynq device over AXI, using ARM/Linux, MicroBlaze, Vivado HLS, and Verity HLS[1]. Each language is able to both call functions exposed by other components, and expose functions to the rest of the system. We experimentally demonstrate the following properties:

- Fast: 1 million calls/sec+ between software and hardware, 4 million calls/sec between hardware clients.
- Scalable: minimal global AXI traffic, so as we increase clients, we are limited only by AXI bandwidth.
- Compatible: everyone can take part without knowing how other clients are implemented.

## II. Motivation and Goals

Bus-based SoC design makes it easy to connect components at the low-level, with component meta-data making it possible

[1] Available at: www.veritygos.org

to determine which ports can connect to which buses. Within the components there is a wide array of languages available to implement functionality, with both hardware and software providing high-level languages for complex or unimportant tasks, and low-level languages for performance critical tasks.

However, what is missing is the ability for components to easily access *functionality* which is implemented on other components. The shared bus only supports low-level reads and writes, so it is often up to the programmer to perform the sequence of reads, writes, interrupts, and polling that make a given component do anything. Our goal is to bridge that gap, and to make it as easy to access a function in a different component as it is to make a local function call.

Remote Procedure Calls (RPC) are a well-known concept for allowing software to access software functions across a network, but incur a large amount of overhead and latency for each call. Intra-SoC function calls present a different challenge, as the fundamental performance of the interconnect is closer to memory than network, and there are different types of actors within the system. So our design goals for PushPush are:

- Compatibility: allow all components within a system to directly communicate.
- Flexibility: make it easy to move functionality between hardware and software domains, or between different implementations within a domain.
- Efficiency: minimise shared bus traffic and overheads, and ensure it can scale to multiple participating clients.
- Performance: minimise the latency of individual function calls, maximise overall function call throughput.

## III. PushPush Protocol

The PushPush protocol is a mechanism for allowing strongly typed function calls between components connected to a shared bus. In this paper a *component* is an IP core or device which wishes to expose and/or call one or more functions over the shared bus. A *function* is an invocable piece of functionality within a component, with a high-level language-independent specification of the input and output types. We assume that race conditions due to multiple calls to the same function are avoided which can be handled within the type system [7], allowing the protocol to focus on one call at a time. We will first describe the general principles of PushPush, and then look at the requirements of a system for it to support the protocol, demonstrating that existing buses can support it.

### A. Protocol Overview

In PushPush a callable function is represented by its *call endpoint*, or **cep**, which is the address of the function within the bus. The **cep** is writeable by any component within the system, and a function call consists of a write starting at the **cep**. The data written consists of the function arguments, followed by the *trigger*. The caller writes a return endpoint to the trigger, and the transition of the **cep** trigger from zero to non-zero informs that callee that a function has been called.

The *return endpoint*, or **rep**, is the address used to receive the return value once the function has finished. Because the **rep** is used as the trigger value for the function, the callee is notified both that the function has been called, and where the caller is located. The **rep** is globally writeable, so once the called function is finished, the callee will write the return value (if any) to the **rep**, followed by a trigger value. The transition of the **rep** trigger from zero to non-zero indicates to the caller that the function has completed execution.

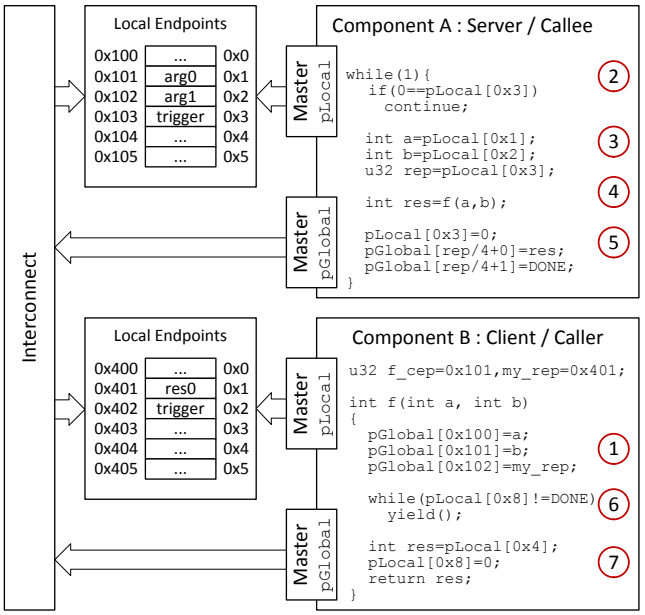


Fig. 1. Interactions between a function exported by component A being imported and called by component B.

Each **cep** and **rep** section finishes with a trigger, and it is up to the component to detect when the trigger transitions from zero to non-zero to detect an event. A component written in RTL can natively detect writes to a slave port, but this is difficult to achieve in most HLS tools without polling, and in software we are restricted to either polling or interrupts. Because all clients already have the ability to poll, we choose to use polling in the default case, but many clients polling a shared memory could easily saturate the shared bus.

Instead, we use the notion of *local endpoints*, which are ranges within the endpoint address space that a component has direct local access to. Each component has its own local endpoint space, which it will use to allocate **cep** s for the functions it exports, and **rep** s for the function calls it makes. Each local endpoint space is still writeable globally, but the associated component also has a direct view onto the local memory, which it can use to very efficiently poll for changes.

An example of a PushPush system is shown in Figure 1, with the upper component exporting a function called *f*, and the lower component importing and calling *f*. Both components are attached to a local endpoint space via a local connection, as well as being attached to the global interconnect. When a function within B calls *f*, the following steps happen:

1) The two parameters *a* and *b* are written to the **cep** of *f* (0x100), followed by a write to the **cep** trigger of *f*. The value written to the trigger is the **rep** of the call endpoint of B (0x401).
2) Component A polls the trigger for *f* at local address 0x3

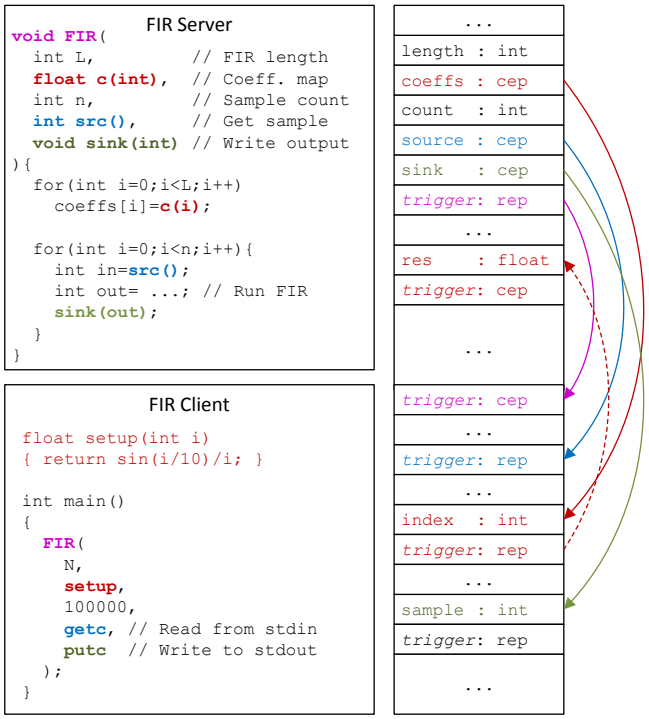


Fig. 2. Endpoint layout and linking of callback functions in an audio filtering example.

(global address 0x103) waiting to receive a call.

3) Once the **cep** trigger has fired, component A unpacks the arguments for the function, and the **rep** of the caller.
4) The target function is called, invoking local hardware or software in the component's language.
5) Once the target function has finished, the return value is written to the **rep** of the caller, then the **rep** trigger is written.
6) Component B has been polling the **rep** trigger, waiting for completion, using only local bus transactions.
7) Once the **rep** trigger fires, the result is unpacked, and returned to the caller within component B.

This protocol offers all the four properties we wish for in the system, as any language which is able to read and write the shared address space can take part. Languages which are able to initiate bursts are particularly fast, but the protocol also supports languages which can only make individual transfers. It also offers good bus efficiency, requiring two transactions per function call in the best (and typical) case, and a number of transactions linear in the argument count in the worst. The efficiency and lack of global polling also ensures it can scale with the number of components, with performance only limited by the bus interconnect.

### B. Parameter marshalling and linking

PushPush relies on the existence of a language-independent specification for function types, which determines the layouts of the **cep** and **rep** segments. Each language also needs a mapping from the language-independent function type to its internal types, with the goal to make this as transparent as possible in most cases. The type system needs to be as general and flexible as possible, to enable maximum expressiveness for designers, while ensuring that all languages can actually expose and consume the described functions.

We rely on the type system introduced in [5], which is inspired by the type system of functional languages such as ML and Haskell. The only types within the systems are functions and primitive types such as integers, booleans and floats, but functions can take as arguments any other type. So as well as taking any number of primitive types, a function can also be passed other functions as an argument.

A function which receives another function is called higher-order, and is found in all modern software languages, including C++11. A key motivation for higher-order programming is to configure functions with other functions, by passing callbacks or primitives to other functions. We will demonstrate both the utility of this approach, and the way that PushPush marshals types, using an audio processing example.

Figure 2 shows two components: the top component exposes a function called *fir*, which performs audio filtering; the bottom component wishes to use the *fir* function. When *fir* is called, it receives both configuration information and inputs and outputs in the form of functions: FIR coefficients are passed using a function which returns the coefficient for each tap index; input audio samples are retrieved by calling a source function; and output audio samples are returned by calling a sink function.

In terms of endpoint layout, we simply allocate one word per parameter (or two in the case of doubles and larger types), in the order they appear in the function signature. The top right of the figure shows the **cep** for *fir*, which simply reflects the parameters of the function type. When a function is passed as an argument, we just write the **cep** address of the function being passed, which is the only information needed to call it.

The arcs from the **cep** parameters to the **cep** addresses show the links established during the function call, allowing the FIR function to access the `stdin` and `stdout` of the calling process, even though the server may be implemented in HLS. The data sources and sinks can be changed by the client, without recompiling the server component, simply by changing the function being passed. A software client can even pass the **cep** of sources and sinks implemented in hardware, allowing software to directly configure and connect two hardware components at run-time.

### C. System Requirements

In order to support the PushPush protocol, we identify five low-level properties that the bus system and bus components must support. The first three are standard features which are supported by all bus systems:

1) Globally routed endpoint address space: The heterogeneous system has a shared and globally routed address space that endpoints exist in. *This does not need to represent the entire address space on a bus, and other address spaces may exist.*
2) Consistent physical endpoint addresses: all components, including software and hardware, have a consistent view of physical addresses with the endpoint space, with each address mapping uniquely to at most one component.

*Outside the endpoint space this may not hold, and there might be aliasing of physical addresses due to locally routed segments. Arbitrary virtual to physical mappings are still allowed within each component.*

3) Reliable ordered interconnect: a write made by one master to a slave will always eventually arrive, and consecutive writes will arrive in order. *This still allows write transactions to be delayed due to arbitration, transactions from different clients may be re-ordered, and bursts can be fragmented or merged.*

We also require two extra features, which are not typically supported, but can be built or achieved automatically in every system we are aware of:

4) Bus masters for all: any component is able to initiate write transactions in the endpoint address space, and so must be able to bus master. *Components not taking part in function calls do not need to have a bus master.*
5) Local endpoint access: each component has a section of endpoint address space that is globally writeable, but which the component also has local read-write access to, without using the global address bus.

Overall these requirements are well within the capability of both the bus systems currently in use, and the languages which are attached to the buses. Unlike network based protocols, it is possible for clients to progressively write arguments, rather than forcing the use of an entire packet. It uses existing bus infrastructure for all endpoint identification and routing, and so can take advantage of existing optimised bus infrastructure.

## IV. Concrete Implementation

The previous section described the PushPush protocol in concrete but general terms. We will now consider the design decisions and details needed to get components written in one language to directly call functions in components written in any other language. The target bus infrastructure is AXI within a Zynq device, and directly support four languages:

- ARM/Linux: C++ code for Linux user-space, with no additional kernel modules, and no mapped interrupts.
- Verity: A higher-order ML-like imperative HLS language, which compiles directly to RTL.
- Micro-blaze/Bare-Metal: C/C++ code written for bare-metal execution, with no extra peripherals or interrupts.
- Vivado-HLS: A first-order C-to-Gates imperative HLS language, which compiles directly to RTL.

There are a number of common decisions and questions needed when providing proxies and stubs to allow these languages to work within the PushPush world:

- Local Endpoints: where are local endpoints placed?
- Notification: how do components detect writes to triggers?
- Client/Server: is the language able to both export and import functions?
- Higher-order: Can the language support functions being passed as arguments for other functions?
- Integration: how late in the compilation process can the existence of PushPush integration be applied?

The overall results are summarised in Table I, and we will now consider each language in turn.

TABLE I
Characteristics of the four language types for integration with the PushPush protocol.

| | ARM | Verity | MBlaze | Vivado-HLS |
|---|---|---|---|---|
| Local Endpoints | OCM | FFs | Block-RAM | Block-RAM |
| Notification | Busy-wait | Write-event | Busy-wait | Busy-wait |
| Client | Yes | | | |
| Server | Yes | | | |
| Higher-order | Full | Full | Partial | Partial |
| Integration | Link | Link | Link | Compile |

### A. ARM/Linux

Zynq devices include a dual-core ARM Cortex-A9 processor, providing enough resources to host an Ubuntu Linux operating system, and a fully-featured environment for high-level software development. The ability to run a full OS is very attractive, so we wanted the ability to expose functions from within Linux to software, both user-supplied and OS-supplied. Another constraint is that proxies and stubs used at run-time should be user-mode only, with no kernel mode components or interrupts, as this is the least invasive form of integration.

For local endpoint storage, the on-chip memory (OCM) is used, as this is local to the ARM Cortex sub-system, and the ARM cores do not cause global traffic when they read and write – however, components on programmable logic still have write access to the OCM. For writes to global endpoints, the global endpoint physical space must be mapped into virtual space, by `mmap`ing `/dev/mem` as root. Notification is implemented by spinning on endpoint triggers, polling them until they change. This is a very wasteful approach in terms of power, however, the busy wait loop executes a `sched_yield` call after checking all pending triggers, so will not hamper other active threads.

The integration of first-order functions into C is direct, allowing any first-order function that does not use pointers to be directly exposed – for example, functions from `libc` such as 'getc' can be directly exposed to hardware components at link-time. C++11 has built-in support for higher-order functions, and software objects using the standard `std::function` wrapper can be exposed as PushPush functions with no modification or re-compilation. Generation of PushPush proxies and stubs is fully automated, and derived from the C++ type information, allowing the generation of support functions at link-time. At run-time these support functions resolve imported **cep** s, allocate any **rep** s, then execute the original code.

### B. MicroBlaze

MicroBlaze is a soft processor distributed by Xilinx, and while it is not as powerful as the ARM, it is ideal for control-intensive management functions such as coordinating other compute-intensive hardware components. Local endpoints are placed in dual-port Block-RAMs; with one port connected to the Local Memory Bus of the MicroBlaze, providing high-speed non-arbitrated access; and the other connected to an AXI slave interface attached to the global bus, enabling writes from other components. Notification is achieved through busy-waiting on the local endpoints, with the MicroBlaze polling the **cep** s and **rep** s in turn. This results in a low latency, growing linearly in proportion to the number of active triggers.

While MicroBlaze supports C++11, it is experimental and relatively high overhead, so restricted higher-order programming is supported, allowing function pointers to be passed or received as arguments. Threading support is also limited, so a co-operative round-robin system is used to detect trigger events. MicroBlaze is well integrated into the Vivado toolchain, so the generation of the MicroBlaze, local memory, and external AXI master and slave ports is automated. When a MicroBlaze component is compiled, the proxy and stub code is generated, then linked with the user's code, and injected into the instruction memory of the IP core. This results in a black-box IP core exporting one or more functions.

### C. Verity

Verity is an imperative HLS, inspired by the ML language family, that can be compiled to synthesisable VHDL [6]. Unlike C-based HLS languages, Verity natively supports high-level language features, such as recursion and higher-order functions. Verity already supports the notion of independently compiled and synthesised functions which are linked together at place-and-route time, but it uses a custom bit-level handshaking protocol derived from game semantics [7]. Importing and exporting Verity functions using PushPush becomes a problem of adapting PushPush to the Verity linker protocol.

Integration is performed by generating a VHDL wrapper for a compiled Verity component, using exposed type information to determine the imported and exported wrappers. AXI-Lite masters and slaves are exposed at the top-level of the wrapper component; AXI-Lite is used mainly for simplicity, and to check that full bursting AXI components can be connected to simpler AXI-Lite components. Local storage is implemented as registers mapped into the local endpoint space of the Verity component. On the Verity side, these registers are adapted to the game semantics protocol spoken by the function.

Because the local endpoints are memory-mapped registers, notification is achieved using logic, which translates writes on the trigger addresses to pulses used in the Verity game semantics protocol. This results in very low latency notification, with only once clock cycle between the trigger being written and the Verity function starting execution.

### D. Vivado HLS

Vivado HLS (VHLS) is a C-to-Gates HLS tools from Xilinx, which allows the creation of hardware components specified in C, C++ and System C. VHLS is well integrated into the AXI eco-system provided by Xilinx, with native support for generating AXI masters and slaves for pointers and RAMs that appear in the code. However, until the recent introduction of SD-Accel, there was no supported route for accessing VHLS functions from software – while it is very easy to connect a VHLS function and the ARM at the bus level, there is a lot of programmer effort involved in actually calling a function. VHLS is also an explicitly first-order language, disallowing function pointers and features such as recursion, so on the face of it is not a good candidate for integration with PushPush.

VHLS has direct support for exposing variables via an AXI slave port, which could be used to support local endpoints, but we have found it more efficient to use the same approach as MicroBlaze. A dual-port block-RAM is associated with the component, with one port exposed via a local pointer to VHLS, and the other connected to an AXI slave adaptor. Global write accesses can be performed using *memcpy* within the VHLS code, which results in high-performance bursts during function calls. Notification is achieved with busy-waits, which is actually efficient in VHLS – the state-machine polls the triggers in local-memory, essentially pausing itself.

System integration with VHLS is a little more complex than the other languages, as it is difficult to directly link a "plain" VHLS core, (i.e. one that is not aware of PushPush). For first-order functions this is possible, as the function type can be extracted, and proxies or stubs built, but this adds a layer of indirection. Higher-order cannot be supported in this way, due to the limitations of the VHLS compiler.

However, it is possible to provide good system integration if a piece of plain VHLS code is compiled alongside a VHLS proxy and stub at IP core creation time. The use of C++ templates and functors within VHLS allows almost perfect proxies and stubs to be created, as long as the C++ front-end is able to statically resolve and expand each function call into the underlying pattern of memory reads and writes. The C++11 standard library object `std::function` cannot be used here, as it internally uses type-erasure which cannot be statically analysed. Hence it is possible to create drop-in replacement functors specialised for HLS, which *can* be statically analysed. These have the same signature and behaviour as as `std::function`, but exist in the `pp` name-space. This allows higher-order programming within VHLS, allowing HLS programs to directly call external functions using the standard function call signature.

## V. Evaluation

PushPush proxy/stub generators have been implemented for all four languages, allowing them to call each other's functions via the PushPush protocol. For initial experiments and correctness testing the Zynq zc702 board is used, containing a xc7z020 clg484 device. Xilinx Vivado 14.3 is used for MicroBlaze compilation, VHLS compilation, synthesis, and place-and-route. ARM software compilation uses g++ 4.9.2, and runs in the Linaro 12.11 distribution using both Zynq CPUs. Verity was compiled using the Nov 2014 development release[2]. The ARM CPUs were clocked at 800MHz for all experiments, and a standard clock constraint of 100MHz was used for all programmable logic and buses.

### A. Function call performance

The main questions are:

- Can the different languages call each other?
- Can each language act as both client and server?
- How much overhead is there per function call?

We test this by performing a deceptively simple experiment: a program is written in each language to either act as a client (importer or caller) of a function of type `int f(int)`, or

[2]commit:`a524fb4d966b89516c5a2047f1af2b263e4e7c4b`.

TABLE II
MEASURED PERFORMANCE IN MCALLS/SEC BETWEEN PAIRS OF CLIENTS AND SERVERS RUNNING A TIGHT DEPENDENCY LOOP.

| | | Server (callee) | | | |
|---|---|---|---|---|---|
| | | ARM | MBlaze | VHLS | Verity |
| Client (caller) | ARM | 3.10 | 2.22 | 2.48 | 1.85 |
| | MBlaze | 1.35 | 1.14 | 2.12 | 2.08 |
| | VHLS | 1.58 | 1.76 | 4.54 | 3.33 |
| | Verity | 1.43 | 2.85 | 3.84 | 4.16 |

to act as a server (exporter or callee). The server performs a stateful update when `f` is called, allowing us to check that the function has actually been called, and the server repeatedly passes the output of `f` back as the input. This ensures there is a dependency chain running through the client and the server via `f`, so the call can not be optimised away or otherwise elided by the compilers in either component.

Table II summarises the results of this experiment using the metric *millions of function calls per second* (MCalls/sec). The first thing to notice is that every entry is positive, meaning every language is able to act as both function importer and exporter, and every language can interoperate with every other language. The second thing to notice is that every entry is above 1 MCall/sec, so the performance is "good" in all cases – recall that this is the performance of a single thread of sequentially dependent function invocations, rather than the number of function calls across many threads.

We can calibrate the performance a little by recalling that the hardware components and bus are running at 100MHz. So for example, a VHLS client can achieve 1.58 MCalls/sec when talking to an ARM function, meaning it takes 63 cycles per call. This includes all time needed for: the VHLS component to initiate the function; the **cep** global write; the Linux user process to detect and evaluate the function call; the **rep** global write; and the VHLS component to detect completion.

Function calls involving software are naturally slower than hardware to hardware calls. Exposing functions from an ARM server to hardware clients is amongst the slowest, as it takes time for the ARM to detect the function call. However, the performance is still high, sustaining a transfer rate of 10.8 MB/sec in terms of data-rate. Hardware to hardware performance is much higher, as HLS languages are quicker at detecting writes to the trigger. When two VHLS components communicate, a sustained performance of 4.54 MCalls/sec is achieved. This means just 22 cycles are used for the entire end-to-end function call, 14 of which are required to enforce the PushPush protocol over the AXI bus, while the remaining 8 cycles are required by the caller and the callee.

A key observation is that the grid of performance does not represent static connections: the system is fully dynamic. From cycle to cycle any of the calls can swap between servers, with no recompilation or reconfiguration. The clients need only call a different **cep**, and they will be calling a different server.

### B. Bus Scaling

PushPush is designed to allow multiple components to co-operate on a bus, and to provide performance that can scale with the number of components. FPGAs do not have a hardened bus infra-structure, so a "bus" is actually an arbitrated interconnect or crossbar, and the logic overhead and performance will change with the number of components. So as we increase the number of components, three questions are:

- How does the observed performance of any individual component change?
- How does the total achieved performance across all components change?
- What is the area overhead?

To investigate these questions we use the same function as before, but limit ourselves to the VHLS component, as this is the fastest and will most stress the interconnect.

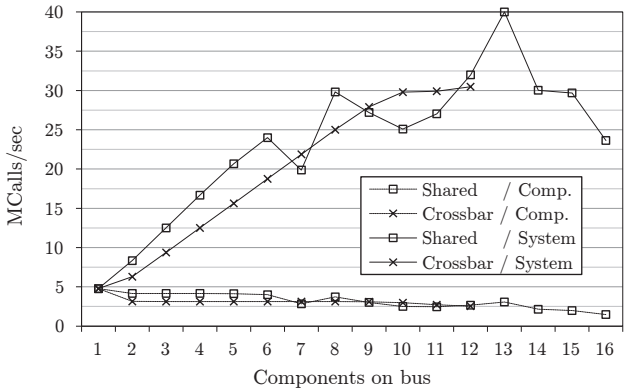


Fig. 3. Change in observed functions calls/sec for Vivado HLS components communicating across a shared AXI interconnect.

As the number of VHLS components in the system is scaled, the number of ports in the AXI interconnect scales at twice the rate. Recall that each component must have a master for writing to other endpoints, and a slave for receiving writes to its local endpoints, so for $n$ components the interconnect will require $n$ masters and $n$ slaves. The AXI infrastructure offers two speed/resource tradeoffs: a shared and arbitrated interconnect, where only one master can be active at any time; and a crossbar interconnect, allowing all masters to be active as long as they are accessing different slaves.

Figure 3 shows the observed performance in the Zynq device, with the upper two lines showing aggregate system performance, and the lower two showing performance per component. Moving from 1 to 2 components shows a slight drop in per component performance, as the interconnect changes from point-to-point to a registered switch. The drop is larger for the crossbar, as it introduces more cycles of latency for each transaction. From $n = 2..6$ the performance is consistent for both shared and crossbar, with linear scaling in total system performance.

At $n = 7$ the different components start to interfere with each other, and arbitration starts to decrease performance and reduce overhead. Performance still increases overall, but is no longer linear. The crossbar scales linearly and consistently up to $n = 9$, then starts to drop off, as the arbitration logic to detect whether masters are accessing independent clients comes into effect.

The crossbar is unable to achieve $n > 13$, as device resources are exceeded. Figure 4 measures the PushPush communication

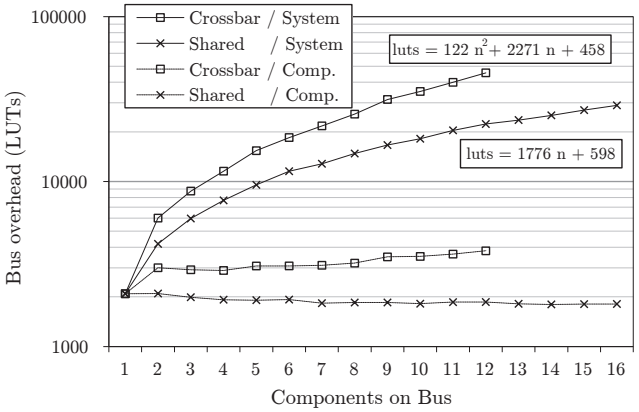


Fig. 4. Resource overhead for AXI communication infrastructure as component count increases.

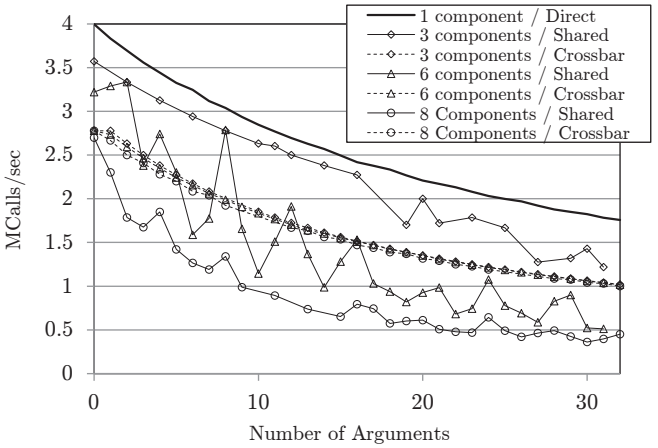


Fig. 5. Observed functions calls/sec for Vivado HLS components exposing functions with different numbers of arguments.

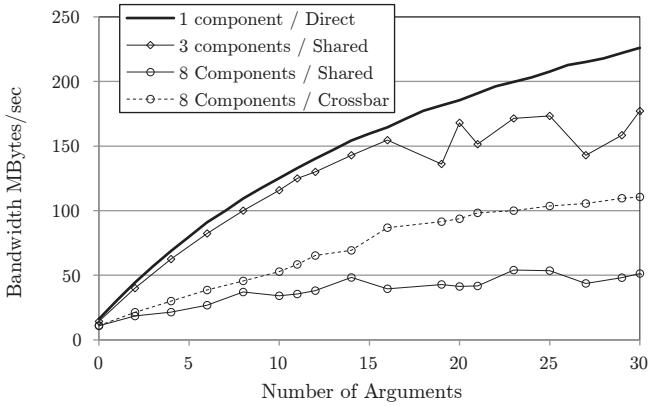


Fig. 6. Achieved data-only bandwidth for Vivado HLS components.

overhead, including the per-component logic needed to support the local endpoints and the AXI interconnect. The resource costs are linear for the shared interconnect, and quadratic for the crossbar, both with $R^2 > 0.99$. The large cost of the crossbar means it is not feasible for highly connected systems, but the performance results show that the lower resource shared interconnect achieves higher performance (though it should be noted that the test-case is somewhat pathological in terms of communication frequency versus payload size).

### C. Argument Size Scaling

PushPush supports any number of primitive or function arguments, so the function signature determines the size of the function call burst write. Functions with more arguments transfer more data per function call, but will have a lower rate, so we now investigate the questions:

- How does the number of arguments affect call rate (MCalls/sec)?
- What data bandwidth can be achieved via functions calls?

To investigate this aspect we modify the VHLS component scaling experiment to functions of type `int f(int, int, ...)`, which accept $n$ integers as parameters. Component scaling is also investigated, with 1, 3, 6, and 8 components, and either direct connection (for 1 component), shared interconnect, or crossbar.

Figure 5 shows the performance in terms of per-component MCalls/sec as the number of parameters is varied from 0 (i.e., a function with no arguments, which can be activated by writing the trigger) to 32. As the argument count increases, the rate slowly drops off as expected – as the argument count grows the function call overhead drops, and data transfer comes to dominate. For small numbers of components the shared interconnect is faster due to its lower latency. However, the crossbar offers much more consistent performance, regardless of the component count, and still achieves more than 1 MCalls/sec for 30+ arguments in a system with 8 components.

Decreasing call rate is compensated by the increasing amount of data transferred per function call. Figure 6 considers the same experiment in terms of the bandwidth achieved, considering only the data payload (input arguments and return value) of the functions, and ignoring control overhead (the triggers). The initial bandwidth is very low, as there is almost no data per function call, but the increasing argument count outstrips the decreasing call rate, so rises quickly. The shared interconnect can achieve around 150 MB/sec for small numbers of components, but arbitration overheads limit it to 50MB/sec for more components; the crossbar is more effective for large systems. Note that these bandwidths are not using streaming, or read-ahead – there is still a dependency chain, so the previous function must complete before the next starts.

## VI. Related Work

Existing methods for connecting components together at at a functional level can be broadly classified as:

- Vertical Software to Hardware: provide tight integration between software and hardware components developed in a shared eco-system.
- Untyped transports: these systems attempt to solve the naming and transport problem, making it easier to move data and notifications between components, but rely on developers to impose meaning and semantics.
- System-level linking: frameworks for binding together functions at compile time, by resolving imported and exported functions and injecting proxies and stubs.

Vertical integration currently provides the closest integration between software and hardware components, as developers can tightly couple the two parts at compile-time. Transparent migration of software components to hardware is provided by LegUp, which creates software proxies and hardware stubs to connect and maintain source level compatibility [2]. Our approach extends LegUp by allowing complete connectivity across all components of the system, while achieving comparable results in software to hardware communication performance (see Table III). Altera's OpenCL compiler is similarly automated, though the programmer use of hardware functions is explicit in OpenCL [1].

Untyped data transport layers aim to act as middle layer, providing a way to move data between components in software and hardware, but stopping short of defining types and interactions. An early example is the Celoxica DataStream Manager [8], which provided a user-configurable set of bi-directional channels between software (PowerPC in Virtex-II Pro or x86 on attached host) and HLS components, abstracting away the PCIe or CoreConnect details. More recent examples include XillyBus [9], LEAP [3], and RIFFA [4], all of which provide some mechanism for reliably moving data between components in hardware and/or software, but largely rely on the programmer to perform explicit function linking. Moreover, it is up to the designer to impose higher level semantics, as none of these approaches natively supports type checking and higher-order function calls. Although the aim of these approaches is different from PushPush as they focus on performance rather than functional aspects, some of them may be used as bridging mechanisms for our protocol.

An approach which provides both compatibility and structure is the notion of a system level linker, which takes independent components with compatible function types, and links them together at system integration time or run-time. One approach, which forms the underlying motivation for this work, is to define a shared system of function types which both software and hardware components can support, then map each function argument to a location in a shared memory [5]. This approach has good compatibility, but poor performance and scalability, as components continually poll across a shared bus, increasing the latency of each function call, and severely limiting scalability as the number of components grow.

Alternative approaches to system linking include the recent Connectal system, which supports the linking of software functions to independently written BlueSpec functions [10]. This provides some of the goals of the PushPush protocol, such as independent development of clients and servers and hardware calling software, but remote function calls are not transparent: the location of functions must be known, proxies must be manually instantiated, and they cannot easily be changed at run-time. Our protocol overcomes these limitations, while achieving higher call rates (see Table III). A distributed systems inspired approach is taken in [11], which builds a complete brokered RPC environment within an SoC. This represents a level above the PushPush layer, with an emphasis on function discovery and name resolution, and is complementary, as the protocol proposed here could be used to optimise the function-call part of their intra-SoC RPC approach.

TABLE III
PERFORMANCE COMPARISON (MCALLS/SEC)

| | LegUp [2] | Connectal [10] | PushPush |
|---|---|---|---|
| HW → HW | - | - | 4.54 |
| HW → SW | - | 0.66 | 1.58 |
| SW → HW | 2.25 | 1.25 | 2.48 |

## VII. CONCLUSION

This paper has presented a protocol called PushPush, which provides an efficient and scalable way of allowing components attached to a shared bus to call functions in other components. The protocol is designed around the notion of pushing both input data and control-flow to the function callee through a single burst write to a call endpoint (**cep**), then pushing the results and control-flow back to the caller via another burst write to a return endpoint (**rep**). The **rep** and **cep** are globally writeable, but are also locally accessible to their associated components, allowing efficient polling to detect function calls.

The protocol is designed to allow any language that can read and write a RAM to take part, both as an importer and caller of functions, and as an exporter of functions that can be called. It also allows native and efficient support of higher-order functions, allowing functions to be passed as arguments just like primitive numeric types.

The compatibility, scalability, and performance of PushPush are proved on a Zynq device, where interoperability between four different languages is demonstrated. Performance in excess of 1 million calls/sec is seen between all languages, whether from software to hardware or hardware to software. Performance between hardware components is even higher, exceeding 4 million calls/sec between two Vivado HLS components, with a total remote function call time of 22 cycles.

## REFERENCES


[1] D. P. Singh, T. S. Czajkowski, and A. Ling, "Harnessing the power of FPGAs using Altera's OpenCL compiler," in *Proc. FPGA*, pp. 5–6, 2013.

[2] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. D. Brown, and J. H. Anderson, "LegUp: An open-source high-level synthesis tool for FPGA-based processor/accelerator systems," *ACM Trans. Embed. Comput. Syst.*, vol. 13, no. 2, pp. 24:1–24:27, Sep. 2013.

[3] M. Adler, K. Fleming, A. Parashar, M. Pellauer, and J. S. Emer, "LEAP scratchpads: automatic memory and cache management for reconfigurable logic," in *Proc. FPGA*, pp. 25–28, 2011.

[4] M. Jacobsen and R. Kastner, "RIFFA 2.0: A reusable integration framework for FPGA accelerators," in *Proc. FPL*, pp. 1–8, 2013.

[5] D. B. Thomas, S. T. Fleming, G. A. Constantinides, and D. R. Ghica, "Transparent linking of compiled software and synthesized hardware," in *Proc. DATE*, pp. 1084–1089, 2015.

[6] E. Aguilar-Pelaez, S. Bayliss, A. Smith, F. Winterstein, D. R. Ghica, D. Thomas, and G. A. Constantinides, "Compiling higher order functional programs to composable digital hardware," in *Proc. FCCM*, p. 234, 2014.

[7] D. Ghica, "Function interface models for hardware compilation,", in *Proc. MEMOCODE*, pp. 131–142, 2011.

[8] J. Jussel, "Software-compiled system design: A methodology for field-programmable design," in *Workshop on Electronic Design Processes*, 2003.

[9] X. Ltd., "An FPGA IP core for easy DMA over PCIe with windows and linux," http://xillybus.com/, 2015.

[10] M. King, J. Hicks, and J. Ankcorn, "Software-driven hardware development," in *Proc. FPGA*, pp. 13–22, 2015.

[11] J. Barba, F. Rincón, F. Moya, J. D. Dondo, and J. C. López, "A comprehensive integration infrastructure for embedded system design," *Microprocessors and Microsystems*, vol. 36, no. 5, pp. 383 – 392, 2012.