Transparent linking of compiled software and synthesized hardware

David B. Thomas*, Shane T. Fleming*, George A. Constantinides* and Dan R. Ghica†

*Dept. of Electrical and Electronic Engineering
Imperial College London, UK
{dt10,sf306,gac1}@imperial.ac.uk

†School of Computer Science
University of Birmingham, UK
d.r.ghica@cs.bham.ac.uk

***Abstract*—Modern heterogeneous devices contain tightly coupled CPU and FPGA logic, allowing low latency access to accelerators. However, designers of the system need to treat accelerated functions specially, with device specific code for instantiating, configuring, and executing accelerators. We present a system level linker, which allows functions in hardware and software to be linked together to create heterogeneous systems. The linker works with post-compilation and post-synthesis components, allowing the designer to transparently move functions between devices simply by linking in either hardware or software object files. The linker places no special emphasis on the software, allowing computation to be initiated from within hardware, with function calls to software to provide services such as file access. A strong type-system ensures that individual code artifacts can be written using the conventions of that domain (C, HLS, VHDL), while allowing direct and transparent linking.**

I. Introduction

High Level Synthesis (HLS) tools have reduced the complexity of hardware circuit design, allowing the automatic creation of accelerated functions [3]. HLS is particularly useful in the context of the trend towards heterogeneous devices containing tightly coupled CPUs and FPGA logic, such as the Xilinx Zynq family of devices. Accelerators created using HLS can be placed in the FPGA, with low-latency communication to the CPU, which could allow hardware and software functions to be interleaved at a finer granularity.

However, while complexity in this area has decreased, a new source of complexity has arisen in system integration. Future systems will contain many heterogeneous compute resources running software and hardware IP from first and third party sources. Interfacing such sources is time-consuming and manual, requiring the integrator to understand both the IP interfaces, and low-level knowledge of the target system.

This paper presents PushPush, a transparent system-level linker for Zynq. PushPush links together pre-compiled modules from different compute domains and languages, called *system objects*, as easily as linking software objects. System objects can be software functions or classes, including binary-only third party libraries; or IP cores in hardware, whether HLS derived, traditional RTL, or post-synthesis black-boxes. System objects (whether software or hardware) are modules which export a particular set of functions for others to use, such as an FFT, and may also import functions that they need to use, such as a "main" function which relies on an FFT.

Since system objects are agnostic to each others implementation and location, PushPush allows function calls to be seamlessly migrated between heterogeneous resources. This allows developers to mix and match system objects and different resources without being required to modify or recompile the objects, as not only can system objects be transparently linked, but swapped or moved at run time. It also supports features common in contemporary software languages, such as callbacks and notifications, using the low latency links between the CPU and FPGA logic to support callbacks from software to hardware lasting microseconds or less.

To enable flexibility while guaranteeing robustness, a strong higher order *type-system for systems*, is required along with an interface protocol accessible to all types of compute device. In this paper we present the type-system and protocol that allowed us to develop a type-safe system level linker, which supporting function calls between heterogeneous system objects.

PushPush is fully operational on the Xilinx Zynq platform, allowing transparent linking between software objects (C/C++) and hardware objects (Verity). As it stands, system objects can only be statically linked at compile time, however it has been designed to support dynamic run-time linking. We present this work to support our manifesto that a system design environment should have:

- **Type-safety** - functions should be strongly typed, and checkable during linking.
- **Language independence** - allow functions written in one language to be called from many other languages.
- **Implementation independence** - functions do not rely on how or where other functions are implemented.
- **True-peering** - function calls can freely cross from hardware to software and vice-versa.
- **Higher order** - supports the callbacks and inversion of control found in contemporary software languages.
- **Automated** - users only job is to specify which hardware and software modules should be linked together.

This paper postulates the importance of such a system design environment, outlining the current progress and details of the PushPush linker. It will cover; exploring the type system in Section III; the abstract protocols required for linking in Section IV; the concrete implementation of the protocol for Zynq in Section V; and some case study applications in Section VI.

TABLE I. COMPARISON OF SYSTEM LINKER FEATURES AGAINST OTHER APPROACHES.

	This	RPC	SWIG	LEAP	LegUp	IRES	Xillybus	Riffa
Type-safe	✓	✓	✓		✓			
Lang. indep.	✓	✓	✓	✓		unknown	✓	✓
Impl. indep.	✓	✓	✓	✓	✓	✓	✓	✓
True-peering	✓			✓		✓		
Higher order	✓							
Automated	✓				✓			
Streaming							✓	✓

II. RELATED APPROACHES

The goals and features of the PushPush can be broadly compared to two types of systems: software-oriented interface and Remote Procedure Call (RPC) generators for inter-language interoperability and distributed systems; and hardware-oriented frameworks for allowing software clients to access hardware accelerators over PCIe and other buses. We summarise the capabilities of these systems relative to PushPush in Table I.

A software oriented interface generator with similar goals is SWIG (Simplified Wrapper and Interface Generator), which parses C++ headers and generates wrappers to expose the functions to languages such as Python, Tcl and Javascript [4]. SWIG focusses on in-process interfaces, but RPC systems can connect functions on different computers via networks, using a shared interface definition file to control marshalling of function parameters. RPC has some of the same aims as PushPush, but requires more manual effort to connect together nodes, and does not support hardware nodes.

One approach to manage hardware-software calls is to provide a generic platform-independent abstraction, such as streams of data, with a platform-specific run-time providing the concrete implementations, such as LEAP [1]. Our approach is complementary to such systems, as while it currently uses its own AXI based transport, it can be layered over other channels. A more vertical approach is taken in systems such as LegUp, which takes a software application and identifies functions to accelerate through benchmarking [3]. Accelerators are compiled using HLS, and the original function call is re-routed to the accelerator. This involves many of the same processes as the system linker, such as the generation of software proxies, but only allows hardware to act as a server and is a single language approach.

Other approaches enable the integration of hardware and software by providing hardware to marshal accesses to instantiated accelerated functions. Xillybus and Riffa [7] provide a hardware bridge that connects to either PCIe or AXI in the software domain and user IP to instantiated FIFOs in the hardware domain. Device drivers are automatically generated for the host platform (Windows/Linux) which can then be accessed through general system calls in the software domain. This is similar for IRES (Integration Linker for Reconfigurable Embedded Systems) [5] where functions are provided to interface with a hardware management unit that marshals accesses to hardware.

III. A TYPE SYSTEM FOR SYSTEMS

To connect functions across compute domains and languages, we have developed a strong type-system influenced by ML and Haskell. The PushPush type system can be mapped to that of other languages provided that they support functions, or have interfaces which can be interpreted as functions, such as hardware accelerators and IP cores.

It has three basic primitives which can be composed together to describe functions and higher order types.

Values (`val`) - Call-by-value; a fixed value passed to a function, such as integer parameters in C.

Expressions (`exp`) - Call-by-name; a value which may change and cause side-effects each time it is evaluated.

Commands (`com`) - Call-by-name; returns no value, but can be evaluated to cause an execution with side-effects.

The arrow operator describes functions; for example, `A->B` where `A` and `B` are primitive types, describes a function that has a input type `A` and a return type `B`. Higher-order functions are ones that either take function arguments inputs or return functions, and are constructed through nesting the -> operator. For example,

`(val->exp)->exp` : is a function with argument of (function) type `val->exp`, producing a result of type `exp` .

`((val->exp)->exp)->exp` : is a function with argument of the previous example type, `(val->exp)->exp`, producing a result of type `exp` .

`(val->exp)->(exp->exp)->com` : is a function with two arguments, the first (function) type `val->exp`, and the second (function) type `exp->exp`; with return type `com`.

Higher-order programming is now common in software development, and built-in support is present in almost all contemporary languages including C++. As a result, modern standard libraries from C++ to Python make extensive use of functions as parameters to configure or modify the behaviour of library functions. Programmers are used to the idea of defining functions to represent tasks and callbacks, then using other functions to co-ordinate and schedule their execution.

We argue that higher-order function are useful for heterogeneous systems, as they provide a simple and flexible way to configure and manage system behaviour. One notable example is debugging FPGA circuits at runtime. Software developers have the ability to insert **print** statements throughout a programs execution and learn about it's internal state. Hardware developers do not have this luxury, even in modern HLS tools, and such functionality is only recently supported in GPUs. However, if hardware developers were to include a **print** (`val->com`) callback in the hardware objects type, then at runtime it can be linked to a software object which can be used to perform the print. This powerful idea gives hardware access to system calls previously unavailable, and because of the low latency coupling in modern reconfigurable system they can be serviced quickly, with sub micro-second overhead for the function call.

The PushPush type system can be mapped onto the type system of any language that support functions, and Table II shows the mappings between abstract PushPush types, C, and C++. This table also illustrates how mappings to modern languages, such as C++11 (available in bare-metal Zynq systems),

TABLE II. MAPPINGS BETWEEN FUNCTIONAL SYNTAX, C, AND C++.

	Abstract	C	C++
1	`val`	int x	int
2	`exp`	int f()	F<int()>
3	`com`	void f()	F<void()>
4	`val -> com`	void f(int a)	F<void(int)>
5	`exp -> com`	void f(int (*a)())	F<void(F<int ()>)>
6	`com -> com`	void f(void (*a)())	F<void(F<void()>)
7	`val->val->exp`	int f(int a, int b)	F<int(int,int)>
8	`val->exp->exp`	int f(int a, int (b*)())	F<int(int, F<int()>)>)>
9	`(val->exp)->exp`	int f(int (*b)(int a))	F<int(F<int(int)>)>

are simpler and more natural than C. The mapping to C++11 makes use of the `std::function` class (shortened to F in Table II), which enables functions to be defined with function types and lambdas/closures for callback functions. To save space, both `val` and `exp` types are displayed in their unrefined form (assumed to be 32 bit ints), however in practice they are refined to include their bit-widths and information on how they should be interpreted (e.g. `float`, `int64_t`).

Lines 1–3 show the PushPush primitive types, with a `val` representing a scope-level constant, while `exp` and `com` are call-by-name functions which may have side-effects. In C++11 `exp` maps to an `std::function` object that contains an `int()` function, as `com` is a `std::function` object that contains a `void()` function. Lines 4 & 5 take one integer argument, but in C differ as to whether it is by value or name. C++11 is different, line 4 is a function object where the argument is a function which takes an input argument, as line 5 is a function object where the argument is another function object. Line 6 consumes a function without any arguments, so the argument is executed as a callback purely for its side-effects. Lines 7 & 8 are both binary functions returning an integer, but in line 9 the brackets turn it into a unary function which takes another unary function as its argument.

We argue that higher-order functions are as beneficial for heterogeneous systems as pure software, as the low latency coupling of FPGA and CPU supports function calls lasting a micro-second or less with little overhead. C++11 higher-order programming is syntactic sugar with no overhead compared to C, and is directly supported by g++ in ARM. Similarly, Python and other languages run perfectly well in Zynq, so some system designers will want to use them and their libraries.

IV. ABSTRACT LINKING OF SYSTEM OBJECTS

To enable the linking of system objects, we define a protocol based around the formal semantics of Game Semantics [6], though the semantics can also be understood at an intuitive level. A call to a function or primitive type is associated with four channels; a question channel (Q), which requests function execution; a parameter channel (P), which supplies arguments to the function; an answer channel (A), for indicating completion; and a data channel (D), for transferring returned data. The parameter and data channels have a width corresponding to the function or type's value, if there are no parameters or return data, then P or D will have zero width. The protocol requires that after a Q event an A event must occur before another Q event, ensuring that each function can only process one question at a time.

System objects can export functions, in which case they act as a server which exposes a Q and P channel for incoming function calls. They can also import functions, in which case they act as clients that send requests to a server's Q and P channel, and expose incoming A and D channels to receive the results of functions they call. A given system object may both export and import functions, so hardware and software objects can act as both client and server, depending on context.

Managing protocol signals in a centralised location would become a point of contention and limit scalability, so PushPush channels are local to each system object, with servers exposing a Q and P channel, and clients exposing A and D channels. During function calls Q and P events are pushed from client to server, then A and D events are pushed back to the client.

System object *endpoints* are the actual locations where the protocol channels reside. For example, system objects in the software domain have endpoints in shared RAM, while system objects in the hardware domain can have endpoints mapped to a bus in the FPGA logic. Mirroring the client and server roles, the endpoints can be divided into two categories: *call endpoints* and *return endpoints*. We will explain these as C-style structs, but they can also be modelled more formally as tuples.

Call endpoints (`cep_struct`) contain the Q and P channels for instigating and executing the target function. Each call endpoint on an object has a unique location, identified by a system-wide *call endpoint ID* (`cep_id_t`). Similarly *return endpoints* (`rep_struct`), contain A and D channels, and are also uniquely identified by *return endpoint IDs* (`rep_id_t`). Figure 1 shows how endpoint structures change as parameters, in particular function callbacks, are passed and returned in a system. Each box represents a separate instantiation of a function packaged as a system object, the top half showing endpoint structures, and pseudo code in the bottom half.

We will now describe the flow of execution in Figure 1, in terms of the labelled points on the figures, starting in `Main`:

1 - The function `f1` has type `(com->exp)->(com)->exp`, so it takes two callback functions as arguments and returns an `exp`. The arguments `f2` and `f3` are passed by name as arguments to `f1`. Each name is a `cep_id_t`, which can identify any call endpoint in the system which matches the parameter type. Here the `cep_id_t` for `f1` is `f1_cep_id` and points to `f1_cep`. The parameter names are sent along the P channel to the argument slots of the `cep_struct`, requiring a strict match between the callback function and the type of the call endpoint. A request on the Q channel for `f1` (`f1_Q`) both starts the execution of `f1`, and uses `rep_id_t` to indicate the return endpoint containing the A and D channels.

2 - The body of `f1` calls function `f2` with `f3` as the input argument. The `cep_id_t`s received as parameters are now used to identify a function to call (`f2_cep_id`), and the parameter to that function (`f3_cep_id`). The only difference between this and the call to `f1` in `main` (**1**) is that one less callback is being passed as an argument, meaning one less `cep_id_t` is required in `f2_cep`.

3 - The function `f3` has type `com`, and is now called as a function with no arguments, and no return value, which requires a single element inside it's `cep_struct` (`f3_cep`) to represent Q (`f3_Q`).

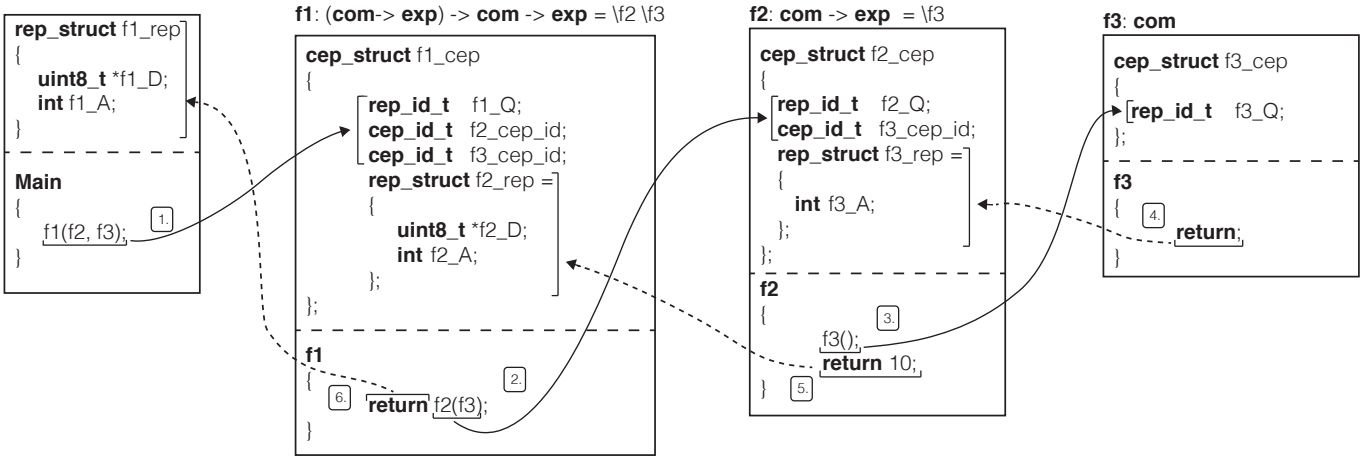

Fig. 1. Example diagram of endpoint structures, where each box represents a system object function. Each function is split into two halves, the top is the endpoint data structure used for the linking protocol, and the bottom represents the functions code

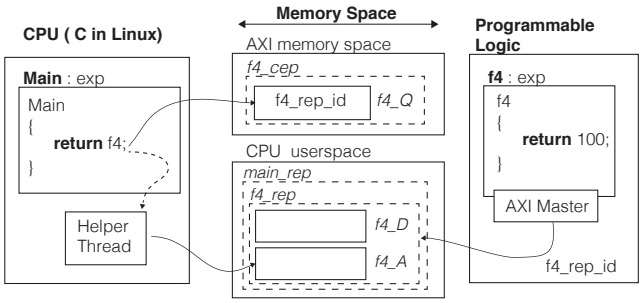

Fig. 2. Simple example aide explaining how a hardware function is linked to software. In this example main is started in software and a call to function f4 is made, an instance of f4 running in the programmable logic is then linked to main.

4 - Once the call to function `f3` has completed, it notifies its caller (`f2`) that the computation has completed using the `rep_id_t` contained in `f3_Q`. Since function `f3` has type `com` no data is returned and the only element within the return endpoint `f3_rep` is the A channel (`f3_A`).

5 - Since function `f2` is of type `com->exp` it has a return type of `exp`. This means that unlike `f3_rep` there is an additional element for data (`f2_D`), which can vary in size based on the returned value.

6 - The return endpoint for function `f1` is very similar to `f2_rep` since it also has an `exp` return type. Once the `f1_A` channel is set the function call is completed and main can continue its execution.

V. Concrete Linking

Linking software and hardware requires concrete interfaces for accessing endpoints. To connect all elements within a Zynq platform we currently use memory mappings over AXI, though it is designed to also support streaming and network channels. In the software domain the endpoints are placed in shared RAM, which can be read and written from the CPU and via AXI. For system objects in the hardware domain, the endpoints are mapped to AXI slave interfaces for accepting data pushed to `cep_struct`, and AXI masters for pushing data to a target `rep_struct`. Endpoint ids are mapped to unique addresses within the global AXI address space. For every hardware object in the system PushPush creates a wrapper instantiating the appropriate AXI slave and master logic and automatically connects them to the wider system.

Figure 2 shows a “main" function executed in the software domain and a function, `f4`, executing in the hardware domain. The call to `f4` requires `f4_cep_id`, allocated by the linker in RAM, and `f4_rep_id`, assigned to a particular AXI address. The server, main, is in the software domain so a helper thread is started and given the location `f4_rep_id` so that it can periodically inspect the A channel of the client (`f4_A`). However, the client `f4` is in the hardware domain, so its `cep_struct` is constructed from registers in the FPGA that are made addressable on the AXI bus via AXI slave logic. The slave logic also checks `f4_Q` for Q events and triggers functions evaluation. A return endpoint, in this case `f4_rep_id`, is sent to the AXI slave register `f4_Q` to provide the location of the return endpoint, where events to the A and D channels `f4_D` and `f4_A`, should be sent.

Once `f4` has completed, its AXI master uses the `rep_id_t` in `f4_Q`, `f4_rep_id`, to send D and A events to `f4_rep`; Finally the helper thread in software detects the assertion of the A channel by the hardware, causing it to continue execution of main.

Currently the linker supports software components written in plain C or C++, and hardware components written in Verity, a higher-order functional language which can be compiled to synthesisable VHDL [2]. The design flow is shown in Figure 3, where the source and compilation flows at the top are pure C/C++ and Verity design flows – no special headers or platform specific APIs need to be included for the code to work with the linker, and it is possible to link against third party compiled code for which the source is not available.

The system linker is shown in gray, and takes as input the software and hardware object files the user wishes to link. First the type information is parsed out of the binary files, using the ELF information in the software binary, or Verity ’s

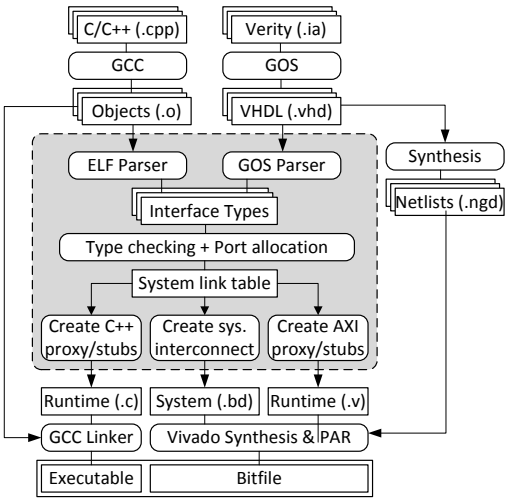

Fig. 3. Design flow, with generic hardware and software toolchains at top, system linker in the middle in grey, and generic platform toolchains at the bottom.

equivalent meta-data. All function names and types are then merged, performing the same checks that a software linker would: all symbols with the same name must have the same type; and any imported symbol must be exported by exactly one module. Violation of these rules results in a link error, just as in a software linker. All symbols are now resolved to one exporter and zero or more importers, and each function parameter and output is assigned a unique location, that can be accessed from both hardware and software.

The final step is to generate linking code, which connects imported symbols in one module to exported symbols in another. Functions imported in software require a proxy with the same name, which translates calls to the proxy to writes on Q and P, and reads on A and D addresses. Functions exported by software require a stub, which listens for requests on the Q channel, and routes the request to the target function. Proxies run on the thread of the user's code, while stubs inherit threads injected by the system linker at program startup. Stubs and proxies are also added on the hardware side, which sit between the AXI bus and the user's components. Write transactions at known Q addresses are translated to Q and P request events for the Verity module, while A and D responses from the module are routed to registers bound to the appropriate addresses.

The final stage is to combine all the components. The software side is linked together with the standard linker, bringing together the user's software and the system linker's proxies and stubs. For hardware, the system linker generates an IP block diagram, incorporating the processing system (CPU), user's hardware, and the AXI proxies and stubs. The result is a single executable, containing both software and hardware. When executed, it will configure the hardware, start any stubs for software functions, then execute `main`, whether that is in hardware or software.

VI. Case Studies

The system linker currently supports the Zynq platform, and can link together ELF software modules with C++ type information against synthesised modules from Verity . Linking against Go and Vivado HLS is supported in a partially-manual alpha form, but here we concentrate on examples using a robust fully automated C++/Verity flow (based on a simpler concrete protocol described in [8]), and focus on qualitative examples of functionality and some limited quantitative results. The case studies used are:

TABLE III. Table showing FPGA area usage and measured execution time in a Zynq ZedBoard running Linux.

	Luts	FFs	Exec. Time	control crossings
stream-avg	1992	3031	10.6ms	6
exp	690	849	1.29ms	1
exp-flip	1155	1188	2.06ms	1
filter	1453	2005	3.59ms	5
sort-cb	2151	519	3589ms	322k
sort	2213	3529	163.8ms	1.6k
sort-par2	3632	5388	20.5ms	1.6k

stream-avg. : Averages a stream of data, with the main function in hardware, making use of IO provided by software.

exp & exp-flip : a single `exp` exposed from hardware to software, or from software to hardware (flipped). This quantifies the underlying cost of the linker startup overhead.

filter : hardware exports a function of type `exp->(exp->exp)->(exp->com)->com`, where the first expression is a data source, the second is a filtering predicate, and the third is a data sink for data matching the predicated; overall there are five domain crossings required during execution to get from hardware back to software.

sort-cb : a single-threaded bubble sorter is exposed by hardware, with the comparison function supplied by the caller, similar to the C library function `qsort`. The software provides 800 elements to be sorted, stressing the AXI protocol due to the 640K calls to the comparison function.

sort : the same setup as *sort*, but now the hardware uses an internal comparison function, ignoring the one supplied by software (software remains the same).

sort-par2 : the hardware exposes the same function type as sort, but now uses two parallel sorts then a merge.

The results in terms of area and performance are shown in Table III, measured on a ZedBoard running Linux. A key main qualitative message is simply that it works – these examples utilise large numbers of cross domain calls, and the system is robust.

Looking at the examples, *exp* and *exp-flip* give an idea of how long it takes to setup and tear-down the linker, which is mainly the cost of starting the stub threads. The marginal cost per call for *filter* and *stream-avg* is lower, as the setup cost is only needed once. The *sort-cb* example demonstrates both a strength and a limitation of the current system: it is possible to use very fine-grain callbacks, with much shorter execution times than is usually efficient, but even so the overhead for very short calls is still significant.

Overall *sort* and *sort-par2* demonstrate our claim that relinking can be used to achieved an area-speed tradeoff. Introducing more parallelism in hardware means the function is faster, but this is a decision the system designer can make at link-time, without needing to modify the software.

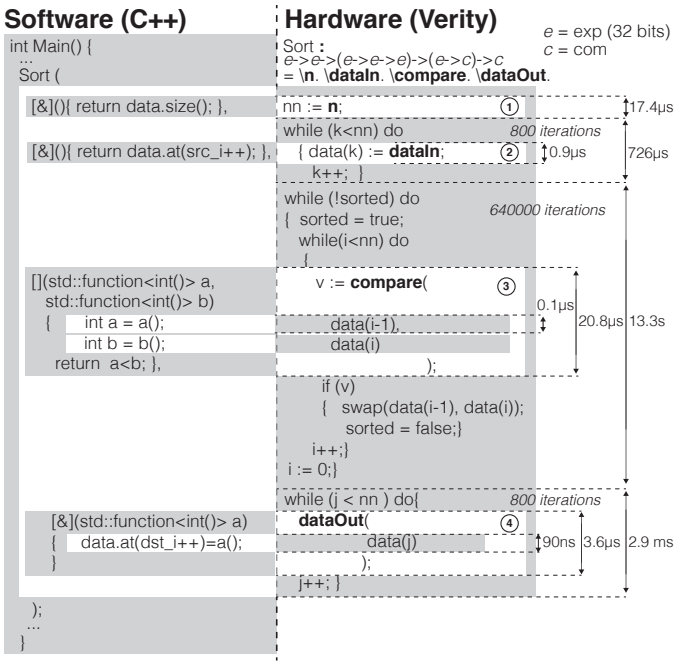

Fig. 4. Diagram showing the control flow for the sorting example. Includes timing for each of the callbacks.

A. Sorting Case Study

Figure 4 breaks down the timing for the iterative bubble sort algorithm, *sort-cb*. The application starts in software (C++11) where a call to a function **sort** is made with four lambda arguments. This call is then linked to a pre-synthesised hardware object (Verity) with an abstract type `exp->exp->(exp->exp->exp)->(exp->com)->com` and four inputs; **n** which returns the size of the list to be sorted; **dataIn** which gets the data from software; **compare** which performs a comparison between two elements; and **dataOut** which writes the data back to software.

1 - First phase of the sort function uses the callback **n**, type `exp`, to execute the first lambda argument it is passed. This callback returns the size of the data set being sorted, and the entire execution time is is $17.4\mu s$.

2 - Callback **dataIn**, type `exp`, is repeatedly used to transfer the data to be sorted from software to hardware. In this example their are 800 elements and the entire transfer takes $726\mu s$, resulting in 1.1 Mcalls/s (millions of calls per second).

3 - The main loop for the bubble sort algorithm, which uses a software function **compare** to compare two neighbouring elements. Compare has type `exp->exp->exp`, in this case both arguments are callbacks, `data(i-1)` and `data(i)`, and both exist in hardware. This means that the line `v := compare(data(i-1),data(i)` causes execution control to flip between software and hardware six times. Overall we achieve 144Kcall/s due to the extra overhead required in creating threads to monitor the inner callbacks.

4 - The final phase calls **dataOut**, type `exp->com`, to write the sorted dataset back to software. The loop containing **dataOut** achieves 278 Kcall/s, which is lower than **dataIn** due to the extra expression that needs to be evaluated and called from software.

VII. Conclusion

This paper has presented PushPush, a transparent system-level linker with the ability to generate linkable system objects regardless of their implementation or language. We demonstrate a fully functional and automated prototype on the Zynq platform and show that by using abstract language-independent types we can seamlessly link hardware and software objects in a type-safe manner. The linking is automatic and functions between sources are linked by name, then type checked. Our type system also supports high-order functions, allowing callback functions as arguments, and providing over 1 million calls/second from hardware to software.

Callbacks enable flexibility in the system design, for example by allowing hardware designers to access operating system calls or `printf` with low overheads. This type of linking enables a more equal partnership between hardware and software, with hardware no longer being a dumb accelerator, and able to initiate execution across the system – we have built systems where "main" is in hardware, and the software only exists to provide functions to hardware. Due to the low-latency of the communication between the FPGA logic and CPU in modern reconfigurable SoCs, and the flexibility provided by PushPush, it is possible to achieve heterogeneous systems with a finer granularity of resource interleaving than before.

Near future work on the PushPush linker will concentrate on increasing the efficiency of the protocol and the scope of the supported languages. Longer-term work will investigate how dynamic linking, such as loading object on demand, and runtime task migration between resources can be achieved. Extensions to the type system will include streaming FIFO type interfaces, and optimised RAM mappings.

References

[1] 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*, pages 25–28, 2011.

[2] 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*, 2014.

[3] Andrew Canis et. al. Legup: An open-source high-level synthesis tool for fpga-based processor/accelerator systems. *ACM Trans. Embed. Comput. Syst.*, 13(2):24:1–24:27, Sept. 2013.

[4] D. M. Beazley. Swig: An easy to use tool for integrating scripting languages with C and C++. In *Proc. USENIX Tcl/Tk Workshop*, TCLTK'96, pages 15–15, Berkeley, CA, USA, 1996. USENIX Association.

[5] J.-C. Chiu, T.-L. Yeh, and M.-K. Leong. The software and hardware integration linker for reconfigurable embedded system. In *Computational Science and Engineering, 2009. CSE'09. International Conference on*, volume 2, pages 520–525. IEEE, 2009.

[6] D. Ghica. Applications of game semantics: From program analysis to hardware synthesis. In *Logic In Computer Science, 2009. LICS '09. 24th Annual IEEE Symposium on*, pages 17–26, Aug 2009.

[7] T. B. Preusser and R. G. Spallek. Ready PCIe data streaming solutions for fpgas. In *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, pages 1–4. IEEE, 2014.

[8] D. B. Thomas, S. T. Fleming, D. R. Ghica, and G. A. Constantinides. System-level linking of synthesised hardware and compiled software using a higher-order type system. In *Proc. FPGA*, 2015.