System-level Linking of Synthesised Hardware and Compiled Software Using a Higher-order Type System

Shane Fleming, David Thomas, George Constantinides
Dept. of Electrical and Electronic Engineering
Imperial College London
{sf306,dt10,gac1}@ic.ac.uk

Dan Ghica
School of Computer Science
University of Birmingham
d.r.ghica@cs.bham.ac.uk

ABSTRACT

Devices with tightly coupled CPUs and FPGA logic allow for the implementation of heterogeneous applications which combine multiple components written in hardware and software languages, including first-party source code and third-party IP. Flexibility in component relationships is important, so that the system designer can move components between software and hardware as the application design evolves. This paper presents a system-level type system and linker, which allows functions in software and hardware components to be directly linked at link time, without requiring any modification or recompilation of the components. The type system is designed to be language agnostic, and exhibits higher-order features, to enables design patterns such as notifications and callbacks to software from within hardware functions. We demonstrate the system through a number of case studies which link compiled software against synthesised hardware in the Xilinx Zynq platform.

Categories and Subject Descriptors

C.0 [**Computer Systems Organization**]: General

Keywords

Co-design, Heterogeneous Systems, HLS

1. INTRODUCTION

The introduction of devices containing tightly coupled CPUs and programmable logic within a single chip have made it possible to physically co-locate an application's software, hardware, and operating system, reducing board design and interconnect complexity. HLS tools allow creation of accelerated functions [2], and there exist reconfigurable operating systems to manage the functions [4], but usually the user must interface software with the accelerators. This process is largely manual, with developers using low-level platform-specific resources or platform independent stream abstractions to move data to and from the accelerator when the function is called [1]. This encourages a client-server mentality, where the software is always in control, and hardware exists to service the software.

In this paper we present a system-level linker, which makes the process of linking together software and hardware components as simple as linking object files together in software. The user can move functions from software to hardware simply by swapping between two components, without modifying or recompiling any of those components. Our key contributions are:

- A minimal type system which captures functions defined in both software and high-level hardware languages, along with an abstract protocol for marshalling function calls.
- A practical system-level linker and tool flow for Zynq, which can combine, type-check, and link heterogeneous systems using a concrete protocol over AXI.
- Case studies demonstrating linking of C++ software and Verity Hardware components in a Zynq device, showing complex control flow including function callbacks, migration of functions between hardware to software by relinking, and inversion of control with the "main" function in hardware controlling software.

The system level linker described in this paper is fully working and automated, but has significant scope for improvement in terms of performance and extra functionality. 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** - user's only job is to specify which hardware and software modules should be linked together.

The system presented here demonstrates that by choosing a common type system, direct hardware to software linking is possible and these goals can be achieved.

FPGA'15, February 22–24, 2015, Monterey, California, USA.

http://dx.doi.org/10.1145/2684746.2689089.

Table 1: Mappings between type system and C.

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

2. A TYPE SYSTEM FOR SYSTEMS

Our type system is influenced by ML and Haskell, and is intended to be mapped to most language type systems supporting functions. There are three primitive 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.

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

The three primitive types can be composed into functions using the arrow operator `A->B`, to describe a function with input type A and output type B. The input and output types can be either primitives, or other functions:

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

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

The first form is a two input first-order function, while the second is a single input higher-order function. The first case is most common, so `->` is defined to be right associative, meaning that `val->(exp->exp)`≡`val->exp->exp`.

The type system is derived from functional programming, but there is a direct mapping to languages such as C and C++, as well as many other common languages. Table 1 shows the mapping between linker types and C. To save space, we show unrefined `val` and `exp` types, which default to 32-bit `int`, but in the real system they are further refined to specify bit-width and how they should be interpreted, such as `float` or `uint64_t`.

Lines 1–3 show the primitive types, with a `val` representing a scope-level constant. The C types for `exp` and `com` show that they are call-by-name, so as functions they may have side-effects. Lines 4 & 5 take one integer argument, but differ in whether it is by value or name. Line 6 consumes a function without any arguments, so 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.

Bindings for other software languages are similarly defined, and in all modern languages the syntax is both simpler and more natural. For example, in C++11 the approach maps naturally onto `std::function` to define arguments with function types, and lambdas/closures to define callback functions. Native support for higher-order functions is present in scripting languages such as Python and Ruby, and modern systems languages such as Go and Rust.

The intrinsic and natural support for higher-order programming in almost all contemporary languages has led to software programmers adopting design patterns exploiting it. Modern standard libraries from C++ to Python make extensive use of functions as parameters to configure or modify the behaviour of library functions. Programmers have becomes used to the idea of defining lambda functions to represent tasks and callbacks, then using other functions to co-ordinate and schedule their execution.

We make the argument that higher-order functions are just as useful in a heterogeneous system, as they provide a simple and flexible way to dynamically configure and manage system behaviour. Higher-order programming in C++11 is often syntactic sugar with little 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 eventually wish to use them.

3. ABSTRACT AND CONCRETE LINKING

At run-time we need a mechanism to allow function calls to be instigated from either hardware or software. The approach we take is adapted from the formal semantics of Verity , but is essentially a simple composable synchronisation protocol that maps well to both hardware and software.

Each primitive (`com`, `exp`, or `val`) is associated with three channels: question (Q), answer (A), and data (D). The Q and A channels carry events, while D transports values – for `exp` and `val` types D has the width of the value, while `com` produces no value, so can be elided. The Q event is written by the consumer (or client) of a value, while the A and D events are written by the producer (or server) of a value.

At an abstract level, the protocol only requires that: for `exp` and `com` , a Q request event must be answered by an A event before any any further Q request; for `exp` the D value must be valid when the A event occurs; for `val` the D value must be valid for the entire scope of a function call.

The exact mapping of these channels to device level constructs depends on where the consumer and producers are located. When both are in software and in the same thread, it becomes a function call: Q is a branch to a function address, D is a value returned in a register or stack, and A is a branch to return address. When both consumer and producer are synthesised together into hardware, Q and A are 1-bit wires, while the D channel is a multi-bit signal which must be valid in the same cycle as the A is signalled. This is how the existing Verity hardware compiler links functions, and can be cheaply translated to the `ap_ack` protocol from Vivado HLS.

To cross from hardware to software we need a concrete protocol to send events and data, and in the current linker, we take an approach supported by every language and compute domain in the system – memory mapping. At link time the primitives representing all function inputs and outputs are collected, then each primitive is mapped to a fixed address. Both producers and consumers share a known address for a particular channel, and memory writes in one compute domain can be eventually observed in all others.

An example of a linked system is shown in Figure 1, where a "main" function written in Verity is linked to two functions written in C. The hardware reads a stream of data from `stdin`, calculates the mean of the data, then prints it to

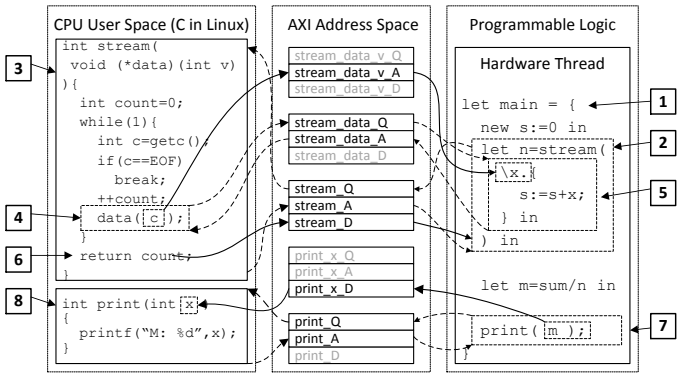

Figure 1: Example of hardware function calling functions in software

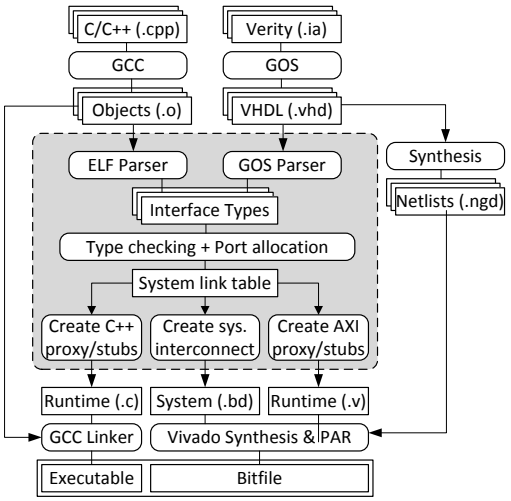

Figure 2: Design flow, with generic hardware and software toolchains at top, system linker in the middle in grey, and generic platform tools at the bottom.

`stdout`. The software `stream` with type `(exp->com)->exp`, using a callback function to push data back to the caller, rather than requiring the caller to pull data. The stages of execution are as follows:

1 - The Verity `main` function begins execution. There are no active user threads in software.

2 - The hardware calls `stream` with a lambda function (`\x.{...}`) as the parameter, by writing to `stream_Q`,

3 - The linker software run-time observes the event on `stream_Q`, and runs the software `stream` function on a worker thread.

4 - When `stream` calls the `data` callback, the runtime passes the argument by value by writing to `stream_data_v_D`, then executes the callback by writing to `stream_data_Q`.

5 - The event on `stream_data_Q` is observed, and routed it to the lambda function in Verity , which updates `sum` as a side-effect. Control is returned by writing to `stream_data_A`.

6 - Once the `stream` function consumes all data, it returns the number of values read by writing to `stream_D`, and signals completion via `stream_A`.

7 - The mean is calculated, then the `print` function is called.

8 - The value is printed in software and control then returns to hardware. At this point the Verity `main` function is finished, and the program halts.

Currently the linker supports software components written in plain C or C++, and hardware components written in Verity , with the design flow shown in Figure 2. The source and compilation flows at the top are pure C/C++ and Verity design flows – no special headers or platform 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 grey, 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, either using the ELF information in the software binary, or Verity 's 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 reads on A 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 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.

4. 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 the robust fully automated C++/Verity flow, and focus on qualitative examples of functionality and some limited quantitative results. The case studies used are:

stream-avg. : The example in Figure 1, with the main function in hardware, using IO exposed by software.

Table 2: 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

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 requests 800 elements to be sorted, stressing the AXI protocol due to the large number of comparisons needed.

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 Figure 2, measured on a ZedBoard running Linux. One main qualitative message is simply that it works – these examples utilise large numbers of cross domain calls, and the system is robust. Even on much larger and more complicated examples the system works, but becomes progressively slower.

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 one limitation of the current system, as while it is possible to use very fine-grain callbacks, the overhead is very high.

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.

5. RELATED WORK

The goals and features of the system linker 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.

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 [3]. 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 our system linker, but requires more manual effort to connect together nodes, and does not support hardware nodes.

Table 3: Comparison of system linker features against other approaches.

	This	RPC	SWIG	LEAP	LegUp
Type-safe	✓	✓	✓		✓
Lang. indep.	✓	✓	✓	✓	
Impl. indep.	✓	✓	✓	✓	✓
True-peering	✓			✓	
Higher order	✓				
Automated	✓				✓

One approach to manage hardware-software calls 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 [2]. 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.

6. CONCLUSION

This paper presents a system-level linker, which uses a language independent type system and abstract protocol to enable direct linking of hardware and software components. Components do not need to know whether they are talking to hardware or software, and can use higher-order functions to support modern programming practices. We demonstrate a fully functional and automated prototype in a Zynq platform, allowing hardware components to link directly to software components running in Linux. Future work will concentrate on increasing the scope of the linker in terms of languages and the optimality of the interconnect.

7. 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] 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.

[3] 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.

[4] E. Lübbers and M. Platzner. Reconos: Multithreaded programming for reconfigurable computers. *ACM Trans. Embed. Comput. Syst.*, 9(1):8:1–8:33, Oct. 2009.