Hume to FPGA

Abdallah Al Zain[1], Wim Vanderbauwhede[2], and Greg Michaelson[3]

[1] Heriot-Watt University, Edinburgh, Scotland, EH14 4AS, UK
A.D.AlZain@hw.ac.uk
[2] University of Glasgow, Glasgow, Scotland, G12 8QQ, UK
wim@dcs.gla.ac.uk
[3] Heriot-Watt University, Edinburgh, Scotland, EH14 4AS, UK
G.Michaelson@hw.ac.uk

Abstract. Hume is a novel language in the functional tradition, strongly oriented to systems requiring strong guarantees that resource bounds are met. To facilitate resource assurance, Hume enforces a separation of coordination and computation concerns, and deploys an abstract machine intermediary between implementations and analyses. These core design decisions also enable a high degree of portability across architectures and suit Hume well to multi-processor implementations. This papers considers how Hume may be implemented on FPGAs via concurrent abstract machines. Initial results from experimental implementations are discussed and the design of a novel FPGA architecture tailored to Hume coordination is presented.

Keywords: FPGA; embedded system; Hume.

1 Introduction

For some little time now, it appears that further advances in single CPU performance have been checked, and that increases in processor speed for commodity platforms are being sought primarily through multiple-cores. Indeed, much is made of soon deploying tens, and indeed hundreds, of cores in one processor, with thread allocation and scheduling controlled transparently by compilers an run-time systems. However, making effective use of multiple cores will still be bound by well known limitations to shared memory multi-processor systems. Essentially, except for very specific algorithms processing very specific patterns of data, shared memory multi-processor performance tails of markedly beyond around 16 processors. It then becomes more effective to build distributed memory assemblies of shared memory nodes, with all the attendant complexities of scheduling and balancing activites, and optimising inter-processor communication, across as well as within nodes.

Field programmable gate arrays (FPGAs) hold considerable promise for avoiding the constraints on von Neumann architectures, by offering the prospect of tailoring low level platforms to higher level algorithms, programs and systems.

However, programming FPGAs in hardware design languages like VHDL or Verilog is notoriously difficult, requiring considerable skill in realising program level constructs directly in hardware level components.

There is much research into using higher order languages for programming FPGAs, often based around extensions to or libraries for functional languages, where the combination of polymorphism and function abstraction simplify the elaboration of high level FPGA configurations and assemblies. While these languages may greatly ease programming, they still have strong low-level orientations, and rely on increasing sophistication in compiler technology to avoid sub-optimal use of FPGA resources.

Recently, there has been considerable interest in using FPGA cores for commodity CPUs as the basis of direct implementations of extant programs. While such cores on a general purpose FPGA necessarily offer worse performance than the equivalent CPU chip, they also offer considerably more flexibility. In particular, contemporary "soft core" architectures such as Xilinx's microBlaze make feasible the assembly and configuration of arbitrary numbers of processors within a single FPGA, without necessarily encountering the constraints of traditional shared and distributed memory von Neumann-based systems.

In this paper we explore a number of different routes to implementing Hume on FPGAs via CPU cores, via the Hume abstract machine (HAM). Our initial experiments were with direct implementations of the HAM on a single Power PC core, for whole program execution. We are now exploring the use of the microBlaze architecture to support the coordination of multiple Hume abstract machines, each concurrently running one or more Hume boxes. In the following sections we survey Hume and its execution model, consider routes from software to FPGA, present our chosen FPGA architecture, and discuss our whole program and concurrent box implementations of Hume.

2 The Hume language

Hume [8] is a functionally-based domain-specific high-level programming language for real-time embedded systems. Hume is designed as a layered language where the *coordination layer* is used to construct reactive systems using a finite-state-automate based notation; while the *expression layer* is used to structure computations using a strict purely functional rule-based notation that maps patterns to expressions. The coordination layer expresses reactive Hume programs as a static system of interconnecting *boxes*. If each box has bounded space cost internally, it follows that the system as a whole also has bounded space cost. Similarly, if each box has bounded time cost, a simple schedulability analysis can be used to determine reaction times to specific inputs, rates of reaction and other important real-time properties.

Moreover, Hume is based on strong semantic underpinnings and is supported by a mature tool-chain centred around the Hume Abstract Machine (HAM). This provides a stable, shared abstraction for both implementation and analysis. The HAM interpreter (Hami) directly executes HAM code, the Hume compiler

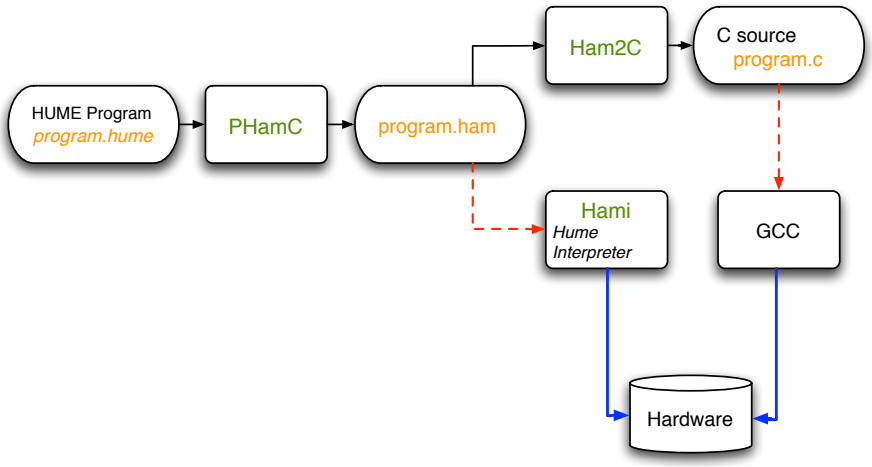

Fig. 1. Hume realisations

(HumeC) translates HAM code to native code through C, and the resource analysers relate statically-inferred properties of HAM to precise instrumentation of equivalent native code on designated hardware platforms, Figure 1.

2.1 Hume Super-step Scheduling

Hume schedules its boxes and wires inside a program using the "super-step" scheduling system. At its simplest, a Hume box consists of a set of *input* wires, a set of *output* wires and a set of *matches*, where each match associates a *pattern* over the inputs with an *expression* over the outputs. As noted, a Hume program then consists of one or more Hume boxes linked to each other, and to the external environment, by wires. The wires thus effectively constitute a shared memory channel between two boxes. Each box may also have additional local memory for inputs, working store, and buffered outputs.

It is important to note that an individual Hume box effectively constitutes an autonomous program that runs continuously, repeatedly matching and consuming inputs to generate new outputs. The Hume semantics specifies a very abstract model of such execution based on a "super-step" scheduling model, which divides program execution into a series of scheduling cycles. Each and every box in the program may be run at most once during each scheduling cycle.

At the start of each super-step scheduling cycle, all boxes are checked to determine whether they are *RUNNABLE*. A box is *RUNNABLE* if it has sufficient inputs to match one of its rules, and it is not currently blocked producing outputs that have not yet been consumed by some other box (in the latter case, it is in a *BLOCKED OUTPUT* state). During the first phase of each scheduling cycle, every *RUNNABLE* box satisfies one of its matches, working on local copies

of its input values to generate locally-buffered output values. It thus ends this phase in the *BLOCKED OUTPUT* state. Note that the order in which boxes are executed in a super-step is arbitrary and immaterial, and that boxes are total, meaning that any *RUNNABLE* box must, by definition, satisfy one of its matches, and be executed during the super-step.

When every box has completed this phase, input wire consumption and output wire instantiation are next resolved globally. First of all, the input values matched by new *BLOCKED OUTPUT* boxes are removed from the corresponding wires. Then, for each *BLOCKED OUTPUT* box, if all its output wires are empty then they are set to the new buffered output values and the box may become *RUNNABLE* in the next cycle. A *BLOCKED OUTPUT* box with one or more non-empty output wire will, however, remain in the *BLOCKED OUTPUT* state.

A useful exception to this tight-stepped scheduling process can be made by distinguishing a *SELF OUTPUT* state, where a box generates outputs solely for its own consumption [6]. So long as a box is *SELF OUTPUT*, it may execute repeatedly without the need for super-step wire resolution. Conversely, not distinguishing such *SELF OUTPUT* boxes may result in other boxes being needlessly scheduled without state change, pending some *SELF OUTPUT* box consuming their inputs or generating their required outputs.

3 From Hume to FPGA

Hume is supported by several different but closely related implementation technologies. First of all, the Hume interpreter written in Haskell is based directly on the formal semantics. While this was a valuable tool during Hume's design and development, it gives poor performance and has been superseded by the Hume compiler family.

The Hume compilers, which share a front end with the reference interpreter, begin with the `phamc` translator from Hume to HAM code. The HAM is a generally familiar functional language abstract machine, which shares many features with those for Haskell and Standard ML. The HAM code is the key locus of Hume resource use analysis, but this is not considered further here.

HAM code may be directly interpreted by the `hami` and its variants, which are written in C. Alternatively, HAM maybe translated to C by the `humec`. The resulting C code may then be compiled to native code, for example by a general purpose compiler such as `gcc`.

From the compiler approach, we have (at least) five different ways to implement Hume on an FPGA. First of all, we might take the native code from the final stage of compilation and run it on the FPGA core for some CPU. While this is an easy route for running Hume sequentially on an FPGA, it brings enormous complications in further FPGA configuration to coordinate multiple boxes on multiple cores.

Secondly, we might translate the C output of the `humec` compiler into an appropriate high level language for direct execution on the FPGA. We have

demonstrated this approach for sequential execution of whole Hume programs. However, adapting it to enable coordinated concurrent box execution would involve considerable intervention in the `humec` compiler to localise code for individual boxes and make explicit either box-to-box coordination or an independent superstep component.

Third, we might design a new FPGA architecture tailored specifically for HAM. This would give optimal performance but at substantial effort, and remains a future goal.

Fourth, we might compile the `hami` to run sequentially on a single core for some CPU. And fifth, we might realise the HAM interpreter as a stand alone FPGA program, again compiling it via some appropriate language to act as an FPGA core for HAM.

The `hami` might seem to offer worse FPGA performance than native code on a CPU core or direct implementation from C. Nonetheless, these routes are relatively low cost and offer promising bases for parallel box implementation, as the `hami` cleanly separates individual box execution from the super step. Furthermore, retaining HAM and the `hami` greatly ease the instrumentation and instantiation of the Hume resource analyses on novel architectures. We explore both of these routes in more detail below.

4 Embedded Systems and FPGAs

4.1 Embedded System

For a simple description, an embedded system is a device that includes a programmable computer but is not itself a general purpose computer, with real-time computing constrains. Embedded systems are hard to define, they comprise nearly all computing systems other than general-purpose computers.

4.2 FPGAs

The invention of *Field Programmable Gate Arrays* (FPGAs), has provided a revolutionary alternative to custom logic chips. FPGAs are versatile configurable electronic devices that can be utilised as accelerators to implement tailored computational logic specific to the application being executed. Moreover, these components can be reconfigured at anytime for new applications, making it possible to perform a wide range of tasks.

The MicroBlaze soft processor core The microBlaze embedded soft core is a reduced instruction set computer (RISC), optimised for Xilinx FPGA implementations [2].

Figure 2 depicts the microBlaze building blocks which include:

- general purpose registers,
- instruction word with three operands and two addressing modes,

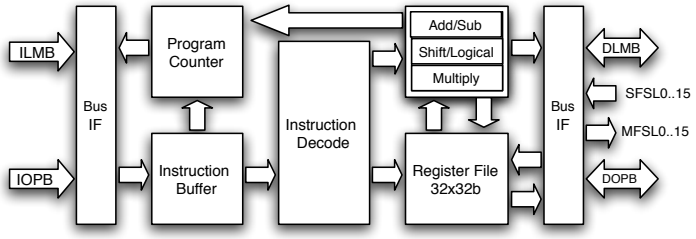

Fig. 2. MicroBlaze Structure

- instructions and data buses that comply with IBM's OPB (On-chip Peripheral Bus) and PLB (Processor Linker Bus) specification, and provide direct connection to on-chip block RAM through LMB (Local Memory Bus),
- instructions to support FSL (Fast Simplex Link),
- and, hardware multiplier.

The microBlaze core implements a Harvard architecture. This means that it has separate bus interface units for data and instruction access. Each bus interface unit is split further into a Local Memory Bus (LMB) and IBM's PLB and OPB buses. The LMB provides single-cycle access to on-chip dual-port block RAM. The PLB and OPB interfaces provide connection to both on and off chip peripherals and memory. The microBlaze core also provides 16 input and 16 output interfaces to FSL (Fast Simplex Link) channels. These are unidirectional, point-to-point non-arbitrated dedicated communication channels.

4.3 FPGAs vs Microprocessors

In the digital world, there are three type of electronic chips: microprocessor, memory, and logic. Memory chips are used to store information. Microprocessors and logic devices, (e.g. FPGAs), are used to manipulate, or interface with the information contained in memory. From a computer architecture prospective, FPGAs are ideally suited to exploit parallel execution. However, that does not imply that FPGAs are better than microprocessor nor that microprocessors substitute FPGAs.

As described in the report by *Mitrionics* [3], FPGAs have a couple of major performance disadvantages compared to microprocessor:

- The maximum clock frequency for FPGAs is a few hundred MHz, while microprocessors run at a few GHz.
- The FPGA's configurability comes at the cost of a large overhead.

Despite these disadvantages FPGAs are still able to outperform microprocessors because:

- FPGAs are used to design specialised circuits for specific tasks.

– All the logic on the FPGA can be utilised to perform the specific task.
– FPGAs deliver a vast amount of fine-grained parallelism.
– FPGAs offer huge memory bandwidths through configurable logic, on-chip block RAMs, and local memories.

5 Hume approaches to FPGA

5.1 Crafting Hume to Run on FPGA

The Hume compiler and Hume interpreter implementations implicity assume the presence of a POSIX-style operating system that takes care of I/O, memory allocation and thread/process-level parallelism. However, an operating system is an unjustifiable overhead in the case of deploying Hume on the FPGA: the FPGa will typically be used as an accelerator on a host system, and the host will take care of I/O; as we will see further, we have no need for file system access, processes or threads either. As a consequence, it was essential to rearrange the Hume implementation of the compiler and the interpreter to be free of OS-specific functionality. For instance, the Hume interpreter relies on the OS timer to coordinate wires, this had to be adjusted to use use library functions which access the processor cycle count registers instead. Moreover, all memory allocations for wires and boxes had to be adapted to a static allocation instead of using *malloc* dynamic assignment. More importantly, embedded hardware systems like FPGA lack the file system concepts, which means Hume implementations had to be enhanced to read input files through dedicated ports on the FPGA board.

All these adjustments to the Hume implementations have been applied carefully to keep the syntax unaltered. By this Hume will continue to provide correct time and space analysis.

5.2 From Hume to FPGA

Figures 3 and 4 illustrate the steps of Hume compiler (*HumeC*) and Hume Interpreter (*Hami*) to FPGA. The figures are revamped from the *Xilinx*'s on-line support documentations [15]. Both figures include the same software and hardware flows. These flows describe the *Xilinx* development tools for microBlaze and powerPC system building process. They include Microprocessor Hardware Specification (MHS) and Microprocessor Software Specification (MSS) files to define hardware and software systems. In our sequential approach from Hume to FPGA, those files has been automatically generated using the *Xilinx* EDK wizard system with minor changes to simplify the memory connection, Local Access Memory (LAM), with the memory controller, Multiple Port Memory Connection (MPMC), to fit with the Hume's wires and boxes design. These hardware and software system files provide the core to build, the microBlaze and powerPC system automatically using *Xilinx* EDK tools. Subsequently, the EDK tools integrate the powerPC and microBlaze cores and the appropriate peripherals, and create custom-built C libraries and drivers. After this, the EDH

uses platform generator and library generator tools, in the hardware and software flows respectively, to setup the particular hardware and software for the corresponding design. In our implementation, platform and library generator tools have to be configured to select our defined STDIN/STDOUT peripherals, map the added peripherals to the appropriate drivers, specify the correct heap and stack size, map the stack and heap to correspondent memory, and set the correct boots and debug options for Hume. At this stage, the *Hami* implementation for FPGA uses the Library Generator to build system-specific library C functions that map the interpreter C functions with the peripherals functions and configure the *C* libraries, Figure 3. For clarification, the Library Generator uses the provided Hume configuration to setup the STDIN/STDOUT for the *Hami* using the STDIN and STDOUT attributes in the MSS file and the *INBYTE* and *OUTBYTE* attributes in the Microprocessor Peripheral Definition (MPD) file. Moreover, the Library Generator writes a *xparameters.h* header file which must be included in the *Hami* header file. The *xparameters.h* file provides essential information for driver function calls and the base addresses of the peripherals in the system. Then the *Hami* source code is compiled to a microBlaze or powerPC binary. The EDK uses the Platform Generator to build the hardware files, which include the system netlists and HDL code and BlockRAM netlists initialised with the program code. These hardware files are then used by the synthesis toolchain to create the final hardware system (i.e. the microBlaze processor and its peripherals) on the FGPA.

The *HumeC* flow to the FPGA, Figure 4, imitates the *Hami* setup as described earlier on. However, the *HumeC* flow produces compiled code for a specific program implementation which gets download onto the FPGA. In contrast to the *Hami* flow which downloads the Hume interpreter to the FPGA and reads the specific program, converted to the *ham* intermediate representation, using the specified port in the peripherals.

6 Hume programs on FPGA

6.1 Hardware Apparatus

In our experiment we used a *Xilinx XUP Virtex-II Pro* Board. It provides an advanced hardware platform that consists of a high performance Virtex-II Pro FPGA surrounded by a comprehensive collection of peripheral components that can be used to create a complex system and to demonstrate the capability of the Virtex-II Pro Platform FPGA. The Virtex-II Pro contains two embedded PowerPC 405 cores and a 10/100 Ethernet PHY device. The board provides up to 2GB of double data rate SDRAM, an RS-232 DB9 serial port, an Ethernet port, up to 256MB of CompactFlash storage, four LEDs and four switches, a 100MHz system clock and a 75MHz SATA clock. It also includes support for FPGA configuration bit-streams.

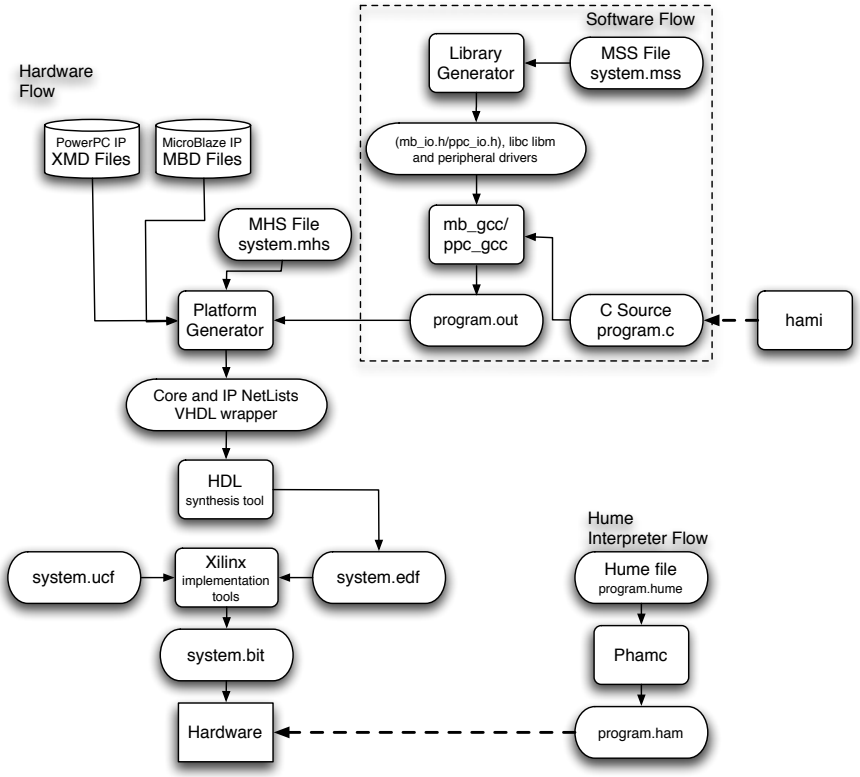

Fig. 3. From Hume Interpreter (Hami) to FPGA

6.2 Software Apparatus

To examine our Hume implementations on the FPGA, we considered two examples, *matrix multiplication* and simple *edge detection*.

Matrix Multiplication The matrix multiplication investigated in our experiment considers the usual pure functional matrix multiplication based on a list representation of matrices. The program consists of one Hume box and multiplies two matrices of 3*3 size.

Edge Detection In this example we considered a simple edge detection algorithm which takes a 240*240 image and runs a standard convolution method. This is done by using a *Gaussian* mask (a 5*5 matrix) and sliding it over every viable pixel in the image. An edge is a sharp difference in the intensity of a pixel and its surrounding elements and the *Gaussian* mask helps to enhance

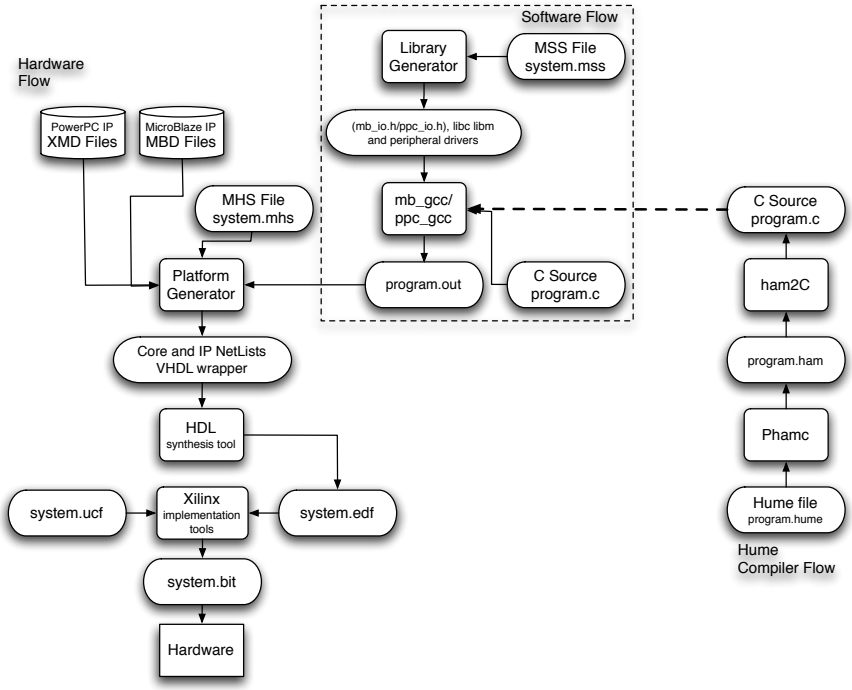

Fig. 4. From Hume compiler code (HumeC) to FPGA

these intensities. The program incorporates six Hume boxes to perform the edge detection evaluation.

7 Evaluation

As we expected, the sequential performance of both examples using *HumeC* and *Hami* on a single microBlaze and powerPC at 100MHz is worse than the performance on an Intel PC with 2.5GHz CPU. This is not a surprise, as we discussed in Section 4.3, but simply follow from the difference in clock frequency. However, on the FPGA we are not limited to a single sequential processor but we can deploy Hume on a large number of cores running in parallel. Furthermore, we can actually tailor the processing core to the Hume language, i.e. create a concrete version of the HAM, and even to the particular program running on each box. Furthermore, because of the flexibility in connecting the cores, we do not face a shared-memory or shared-bus bottleneck. For these reasons we believe that the parallel FPGA is the right way of our Hume Embedded language to follow.

There are two main characteristics of FPGAs that leads us to consider FPGAs over traditional microprocessors, principally for embedded systems:

- real-time operation [13], which requires predictable performance from the architecture. This does not mean that the architecture has to be trivial, such as eliminating caches. It does mean that the architectural elements have to behave predictably enough so that the compiler and programmer can plan how to achieve the required computation rates in critical parts of the system. As we explained in Section 5.2, the FPGA programming tools provide the apparatus to obtain the feasible design for Hume implementation in embedded system. Essentially, the Hume programming language has been designed to enable highly accurate analyses of software component resource use [10, 11].
- low-power and energy operation [16], the most concerned aspects of this to parallel programming are that the architecture's energy and power consumption characteristics must be as predictable as possible. However, in particular in embedded systems, low-power operation is essential to prolong battery life, and due to their low clock speeds, power consumption of FPGAs is an order of magnitude lower than for microprocessors.

8 Future Work

Based on our evaluation discussed on the previous section, we designed a parallel realisation of Hume in the FPGA, Figure 5.

In this design, Hume boxes will be evaluated in parallel since each box will be assigned to a separate microBlaze. FPGA provides communication channels FSL [1], Section 4.2. The FSL channel are dedicated unidirectional point-to-point data streaming interface. All communications between boxes will be through the FSL channels which, we believe, will achieve *completely lock-free communication* between boxes. There is no need to synchronise two communicating boxes: since any output written during one Hume scheduling cycle will never be read before the subsequent cycle, and since the subsequent cycle will not be scheduled before all output is completed, it follows that a box/microBlaze can never start to read data before it has completely finished its output.

9 Related Work

There have been some attempts to extends functional-based programming languages to use FPGAs,

- Lava [4, 5], it extends Haskell with operations that allow the high-level description of FPGA circuits.
- Intel's reFL^{ect} [7]: it is strongly typed and similar to ML, but has quotation and anti-quotation constructs. Its features intended for applications in hardware design and verification.
- MetaML [12], it is very similar to Intel's reFL^{ect} with more direct aim on program generation and control and optimization of evaluation.

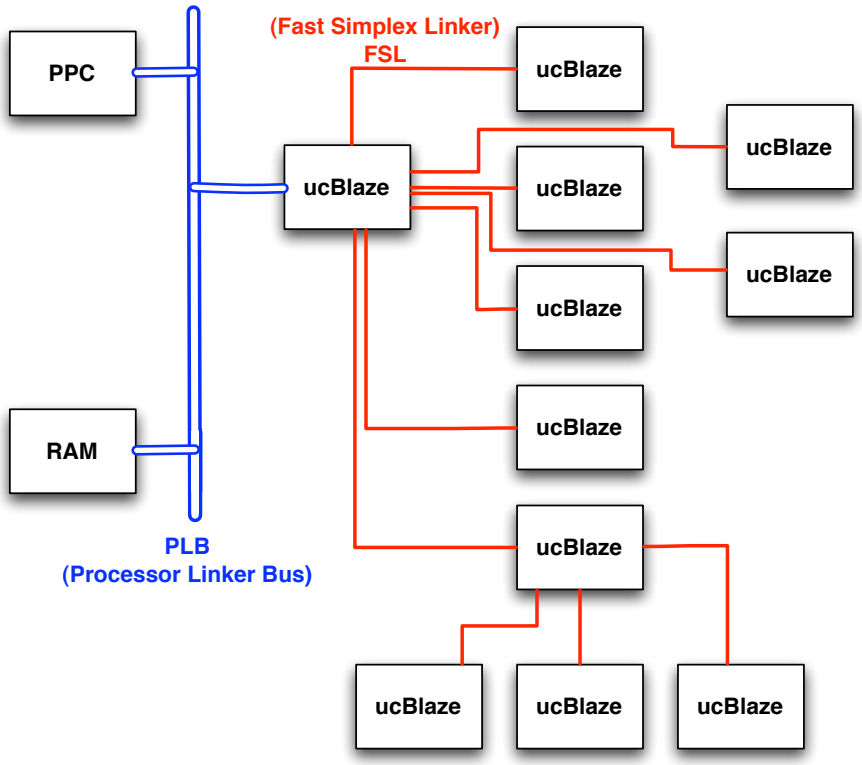

Fig. 5. Balance parallel design for Hume boxes on the FPGA

- Template Haskell [14], it aims at program generation and the control and optimization of evaluation. To support this it generates code at compile time which allows the programmer to implement such features as polytypic programs, macro-like expansion, user directed optimization (such as inlining), and the generation of supporting data structures and functions from existing data structures and functions.
- The functional derivation approach, for deriving FPGA circuits from Haskell specifications [9].

Our work, as presented in this paper, is novel in adopting a soft processor approach and in attempting to follow a complete development path from source language to target FPGA hardware.

10 Conclusion

In this paper, we have presented our motivation for deploying programs written in Hume, a functional language aimed at systems requiring strong guarantees on

resource bounds, onto FPGAs. We have discussed our preliminary experiments and findings and outlined the route we are currently exploring to exploit the FPGA's potential for parallelism, i.e. by creating a network of soft processors which will run Hume programs in a truly concurrent fashion.

Acknowledgements

This research is supported by the UK EPSRC Islay project (EP/F030592, EP/F030657, EP/F03072X, and EP/F031017) "Adaptive Hardware Systems with Novel Algorithmic Design and Guaranteed Resource Bounds".

We would like to thank Kevin Hammond, who designed the HAM and built the original `phamc` and `hami`, and Robert Pointon, who designed and built the `humec`, and further extended the `phamc` and `hami`.

References

1. Fast simplex link. Online Technical Report. http://www.xilinx.com/products/ipcenter/FSL.htm.
2. MicroBlaze Processor Reference Guide (v 4.0). Technical report, Xilinx, 2004.
3. Low power hybrid computing for efficient software acceleration. Technical report, Mitrion TM, 2008.
4. P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: Hardware design in Haskell. In *Proc. of International Conference on Functional Programming (ICFP)*. ACM SIGPLAN, 1998.
5. K. Claessen and G. J. Pace. An embedded language framework for hardware compilation. In *Designing Correct Circuits '02, Grenoble, France*, April 2002.
6. G. Grov, G. Michaelson, and A. Ireland. Formal Verification of Concurrent Scheduling Strategies using TLA. In *3rd IEEE International Workshop on Scheduling and Resource Management for Parallel and Distributed Systems*, number CFP07036-USB in IEEE Catalog Number. IEEE, 2007.
7. J. Grundy, T. Melham, and J. O'leary. A reflective functional language for hardware design and theorem proving. *Journal Functional Programming*, 16(2):157–196, 2006.
8. K. Hammond and G. Michaelson. Hume: a Domain-Specific Language for Real-Time Embedded Systems. In *Proc. GPCE 2003: Intl. Conf. on Generative Prog. and Component Eng., Erfurt, Germany*, pages 37–56. Springer-Verlag LNCS 2830, Sep. 2003.
9. J. Hawkins and A. Abdallah. Behavioural synthesis of a parallel hardware jpeg decoder from a functitonal specification. In *Proc. EuroPar 2002*, August 2002.
10. M. Hofmann and S. Jost. Static Prediction of Heap Space Usage for First-Order Functional Programs. In *Proc. POPL '03: ACM Symp. on Principles of Prog. Langs.*, pages 185–197, Jan. 2003.
11. S. Jost, K. Hammond, H.-W. Loidl, and M. Hofmann. Static determination of quantitative resource usage for higher-order programs. In *POPL '10: Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 223–236, New York, NY, USA, 2010. ACM.

12. E. Pasalic, W. Taha, and T. Sheard. Tagless staged interpreters for typed languages. In *In the International Conference on Functional Programming (ICFP 02*, pages 218–229. ACM, 2002.
13. C. P. Ravikumar. Multiprocessor architectures for embedded system-on-chip applications. In *VLSID '04: Proceedings of the 17th International Conference on VLSI Design*, page 512, Washington, DC, USA, 2004. IEEE Computer Society.
14. T. Sheard and S. Peyton Jones. Template meta-programming for haskell. *SIGPLAN Not.*, 37(12):60–75, 2002.
15. Xilinx support documentation. Online White Papers. http://www.xilinx.com/support/documentation/white_papers/.
16. W. Wolf. The future of multiprocessor systems-on-chips. In *DAC '04: Proceedings of the 41st annual Design Automation Conference*, pages 681–685, New York, NY, USA, 2004. ACM.