A Compute Model for Generating High Performance Computing SoCs on Hybrid Systems with FPGAs

Felix Friedrich, ETH Zurich; Oleksii Morozov, HighDim GmbH; Patrick Hunziker, University Hospital Basel

Abstract

The design of complex systems on FPGAs can be challenging because of the semantic gap between development of high-performance IP blocks on programmable hardware and the construction of software for general purpose processors. We aim at closing this gap between software- and programmable hardware development.
To deliver maximal flexibility and performance while simplifying the design process, we propose a unification of the dissimilar blocks of such hybrid systems. Targeting hybrid CPU-FPGA architectures in this manuscript we provide high-level support for combining and connecting high-performance IP blocks with soft-core processors and prefabricated Systems-on-Chip. Hardware structure and software design are both defined in a unified programming environment. A tool-chain automatically integrates soft-core code compilation, standardized connection of the involved components and custom embedded runtime support, minimizing the number of steps for system development. An important design objective is the simple extensibility for the definition of new IP blocks and the integration of new target hardware. In case studies with various challenges from the medical computing domain, ranging from real-time multi-core medical monitoring to high-performance signal processing, the practical value of our approach in creating high-performance embedded systems is demonstrated.

1 Introduction

While early FPGA programming was left to experts in the field, current research explores how FPGAs can be made accessible to domain experts in different fields by making available tools that automatically map high level algorithm descriptions to highly efficient FPGA circuits. In the recent decade a significant amount of research has been conducted about how FPGAs can be made accessible to domain experts in different fields. Automated tools have emerged that automatically map algorithms described on a high level to highly efficient circuits on FPGA hardware.
However, the performance achieved by current dedicated CPUs, GPUs and DSPs for specific tasks renders their integration with FPGA processing also highly desirable, although at current, the development of such hybrid systems is challenging and requires specific know-how in different engineering domains, in particular if the specific advantages of FPGA, namely flexibility, hardware timed parallelism at various levels and low power for specific computational tasks, are to be fully exploited.
We initially addressed the needs of simple FPGA programming with an approach that connected multiple tiny soft processors in a point-to-point fashion. The corresponding programming model consists of an environment of communicating sequential processes described in a high-level programming language. The resulting dataflow network can be deployed directly from software to reconfigurable hardware: processes are mapped to processors on a custom network on chip. This model fits well to many application fields, in particular to the medical streaming applications envisioned by our consortium.
In order to additionally address performance issues, we added to our initial programming model

1. the provision of hand-crafted components, *engines*, for special tasks such as computing scalar products, performing an FFT, streaming convolutions,
2. the configurability of soft-core processors in terms of features they support, such as floating point unit, vector unit, memory sizes and instruction bit widths,
3. a way to seamlessly integrate hard-wired processors.

Most of our design decisions were driven by the needs of applications. Moreover, in order to keep the programming model simple but expressive, we tried to map as close to the targeted streaming applications as possible, implying the notion of independent processes, a reasonable dynamic way to define interconnect topologies and the transfer of properties from the software source code into the hardware synthesis process.
Our model provides the following benefits:

- Simple design of complex streaming applications in a single environment,
- Unified definition of hard- and software consistently using a single high-level programming language,

- Highly readable code,
- Capability to deliver high performance on systems with FPGAs,
- Support of hybrid systems consisting of general purpose CPUs and FPGAs,
- Short development cycles during development phases when the network interconnect topology remains unchanged,
- Extensible and portable hardware library of custom or third party components
- Portabilitiy and simple specification of new hardware platforms

The system is largely independent, although in the background FGPA vendors synthesis and implementation tools are utilized.
The unified approach to hard- and software development proved its educational value in system construction courses given at ETH Zürich. Moreover, it proved its real-world applicability in productive industrial deployments in health care.
The paper is organized as follows: Section 2 focuses on related research and tools. In Section 3 we present key aspects of programming model relevant to this paper. In Section 4 we describe the compilation process and explore how hybrid compilation can provide shortened development cycle times. Section 5 explains technical aspects of building extensible libraries of components. In Section 6 we illustrate what can be achieved in case studies from the area of medical computing which reflect different real world requirements. Section 7 concludes the paper.

2 Related Work

In this chapter we compare our approach with other high-level programming language tools for FPGA development. The two largest vendors of FPGA Hardware, Xilinx and Altera, provide tools to generate circuits from high level languages. Xilinx' Vivado platform now includes High Level Synthesis (formerly referred to as AutoPilot [39] or AutoESL [8]) from languages such as C++, C or SystemC [30]. Altera builds on OpenCL as the language for high level synthesis [9]. An overview is provided by [7]. The mentioned platforms are versatile but at the same time of significant complexity rendering the understanding of the hardware generation process rather challenging.
While having different roots, our project and the liquid metal (Lime [21] [5]) project at IBM research have converged to certain common features. Lime comprises a programming model based on Java and aims at compiling code to heterogeneous architectures consisting of CPUs and FPGAs. Software is initially compiled to Java bytecode that can run in a Java virtual machine. Additionally there are compiler back-ends available that generate HDL code for various architectures. In order to be able to do static analysis on the task graphs provided with the Java code, the code has to be sufficiently annotated [4]. Stream-graphs can be described in Lime with explicit connection statements. Lime thus combines the facilities of High Level Synthesis with the generation of code for Java virtual machine. Lime does not seem to take into account the possibility to explicitly integrate soft-core processors into the task graph. Our approach is more explicit in the choice of hardware-components and the language support we consider more lightweight. We integrate native code and runtime-systems for the processors involved and do not rely on additional steps in their integration. Our compilation and deployment to hybrid-systems is a one-click action without further configuration or build necessities.
A research project similar to Lime, Kiwi[33], was carried out at Microsoft aiming at automatically translating C# programs to a hardware description language (HDL). In a similar fashion to Lime, byte code is automatically translated into circuits with a hardware synthesis system. The integration of processors in a heterogeneous system is not mentioned.
The following approaches also aim at direct translation from a high level language to HDL but are not expressively supporting hybrid systems: ASC [24] generates HDL code from parameterized C-Code. The Java-like StreamIt language [36] integrated with the Optimus framework [20] for FPGAs focuses on the concepts of streams. It provides an awareness of timing effects in pipelines of filters, an aspect that we are about to integrate into the HDL generator of our compiler. Chisel [6] is a framework that embeds hardware construction primitives using the Scala programming language.
The modern successor of ASC, MaxCompiler [25], is a framework that generates accelerator code for FPGAs written in a specific Java-based language and manages interaction between CPU-code and the accelerator code based on a high-level programming language.
Most approaches that aim at generating circuits from a high level perspective require significant interaction with the programmer during the compilation and deployment process. Very prominently this is true for Matlab or Simulink based approaches (c.f. the seminal paper [16], Matlab applied to Artificial Neural Networks [27], Matlab targeting heterogeneous systems [34]), partially caused by lack of type inferences. We believe that an important property of a tool-chain is its capability to generate a system without further guidance by the programmer.
We are not aware of a system that makes use of the benefits of fast software compilation described in Section 4 below. Moreover, we have not found any similarities on our attempt to provide a lightweight framework for hybrid systems that is easily extensible as described in Section 5 below.
Unlike the authors of [32], we have not made the attempt to automatically parameterize soft-cores from software. We consider soft-cores controlling elements steering the behavior of the high performance engines on the FPGA.

The runtime kernels generated for the hardware CPU blocks are based on the Active Objects kernel [26], and more recently on a lightweight, lock-free portable operating system kernel [28], [29]. We are not relying on a Linux-kernel.

Concluding this section, we would like to comment on the language design: we started with a naive view on the hardware and first looked at programming models for dataflow computing [10], [22], at various process models such as Kahn process networks [23], at the Actor model [18], [2] and at the model of communicating sequential processes [19] and various implementations such as Occam [31], Joice [14], Superpascal [15] and Google's go [12].

The compute model and tool-chain presented in this paper is largely based on the work presented in [11], a programming model that initially had implemented Kahn Process Networks. Driven by the requirements of applications we had to complement the model with nondeterministic features such as asynchronous communication.

[13] describes a very similar programming model form the perspective of Actor models. In order to gain the required flexibility and extensibility described in Section 5 below, we had to improve certain aspects in the programming model and to reimplement the compiler and HDL-generation components.

3 The Underlying Programming Model

The programming model *ActiveCells* [11] has been provided as an extension to the imperative, object oriented, programming language Active Oberon that supports shared-memory concurrency with active objects [26].

The ActiveCells model adds an explicit notion of components that communicate via channels, bringing the programming language close to the model of communicating sequential processes. Following a declarative approach, the model features the explicit construction of distributed concurrent systems (including SoCs).

Figure 1 presents an example implementation of an Artificial Neural Network (ANN) with ActiveCells that we use in order to describe the concepts of our programming model in the following. The following features were added to the language: Cells equipped with communication Ports, Cellnets, connection and delegation, sending and receiving.

Cells are declared as types with an integrated activity that runs in isolation from activities of all other cells. The `Controller` cell provides code that runs on a processor core. In this case, this is an ARM processor of an FPGA-CPU hybrid Zynq system. A cell communicates with other components via ports. Messages can be passed to other cells by writing to an out-port, receiving is done by reading from an in-port, as visible in the body of the `Controller` cell in lines 21 and 23. Writing and reading is blocking by default. Non-blocking variants of send and receive are available, too. Cells can be instantiated as processors determined by the built-in property "Processor", either as soft-core or on a pre-built hardware, or as *engines*. *Engines* are pre-fabricated components written in Hardware Description Language (HDL), custom designed or automatically generated by the HDL backend runtime or using external tools (e.g. Xilinx CORE generator or Vivado HLS).

The crucial building block for an ANN is the perceptron, a function that maps an n-dimensional vector of inputs to a scalar. Figure 1 presents a perceptron that takes as input the vector $\mathbf{x}$ and a weight vector $\mathbf{w}$ and computes the inner product plus bias b as $\sum_i x_i \cdot w_i + b$ and feeds the result to a nonlinear activation function (here sigmoid) that then yields the output of the perceptron. Obviously a perceptron combines elementary actions such as multiplication and addition, and thus can be built from simple multipliers, adders etc. Alternatively, it can be constructed from more high-level, potentially highly optimized, components like inner product or sigmoidal function. The construction of a Perceptron is implemented from line 37 to line 55.

Cellnets provide a way to dynamically set up networks of cells and other cellnets. Like cells, cellnets are declared as types and have an executable body. However, in contrast to cells, cellnets cannot communicate but only instantiate and connect cells. Cells are connected using FIFOs of parameterizable length which are inserted between sender and receiver ports, similar to the concept of pipes in OpenCL [35]. Cellnets that expose ports in their interface can delegate ports, thereby allowing hierarchical cellnet composition. The execution of a cellnet body is synchronous, i.e. it runs together with the instantiation of a cellnet (thus allowing convenient initialization), while the execution of a cell body is asynchronous, i.e. it starts to run after instantiation of a cellnet.

The interfaces of cells and cellnets consist of named ports and properties, such as the number of inputs of the perceptron specified in line 37. The port names serve as reference for connection and communication and can also be referred to by the HDL generator. Properties of a component can be provided at the declaration of a component, the declaration of a component instance or during instantiation. Properties are also passed to the HDL generator and may influence the compilation process. This is made more explicit in the next section.

A *terminal* cellnet is the deployment unit in ActiveCells: it can be instantiated and deployed to FPGA hardware or executed on the development system. The `ExampleAnn` cellnet provides the code for construction of a one-layer ANN driven from software running on the controller core. In real applications ANN have multiple layers, which are easily composable based on the provided example.

When the body of `ExampleAnn` starts from line 65 ,the following events happen, the resulting SoC is displayed in Figure 2: In line 66, a new controller cell `ctl` is allocated and configured to its target, an ARM core of a Zynq7000 hybrid CPU-FPGA system, by setting its `Processor` property to `ZynqPS7`. As `ctl` is a cell, the `new` statement immediately returns. In the following for-loop the layer consisting of multiple perceptrons is generated: new perceptrons are allocated in line 66. During allocation of

```
1   module Ann;
2   import Engines;
3   (* ARM processor *)
4    Controller = cell{Inputs=3,Nodes=3}(
5      x: array Inputs of port out;
6      w, b: array Nodes of port out;
7      y: array Nodes of port in
8    );
9   var
10     wVal: array Inputs, Nodes of real;
11     xVal, yVal: array Nodes of real;
12     i: longint;
13  begin
14     (* data input and output omitted for brevity *)
15     for i := 0 to Inputs-1 do
16         wVal[i] >> w;
17     end;
18     bVal >> b;
19     loop
20        ...
21       xVal >> x; (* send *)
22        ...
23       y >> yVal; (* receive *)
24        ...
25     end;
26  end Controller;
27
28  Sigmoid = cellnet( x: port in; y: port out );
29  var
30     interpolator : Engines.LinearInterpolatorFlt;
31  begin
32     new(interpolator{CoeffFile="Sigmoid.coeff"});
33     x => interpolator.x; (* delegation *)
34     interpolator.y => y;
35  end Sigmoid;
36
37  Perceptron = cellnet{Inputs=3}(
38     x, w: array Inputs of port in;
39     b: port in; y: port out
40  );
41  var
42     dot: Engines.InnerProdFlt;
43     add: Engines.AddFlt;
44     sigmoid: Sigmoid;
45  begin
46     new(dot{Size=Inputs});
47     x => dot.x;
48     w => dot.y;
49     new(add);
50     b => add.input[0];
51     dot.z => add.input[1];
52     new(sigmoid);
53     add.output => sigmoid.x;
54     sigmoid.y => y;
55  end Perceptron;
56
57  ExampleAnn = cellnet; (* terminal cellnet *)
58  const
59     Inputs = 5; Nodes = 7;
60  var
61     ctl{Inputs=Inputs,Nodes=Nodes}: Controller;
62     node{Inputs=Inputs}: Perceptron;
63     reshape{M=1,N=Inputs}: Engines.StreamDeserializer;
64     i: longint;
65  begin
66     new(ctl, {Processor="ZynqPs7"});
67     for i := 0 to Nodes-1 do
68        new(node);
69        new(reshaper);
70        ctl.w[i] => reshaper.input[0];  (* connection *)
71        reshaper.output => node.w;
72        ctl.b[i] => node.b;
73        ctl.x => node.x;
74        node.y => ctl.y[i];
75     end;
76  end ExampleAnn;
77
78  end Ann.
```

Listing 1 Example Implementation of an Artificial Neural Network with ActiveCells

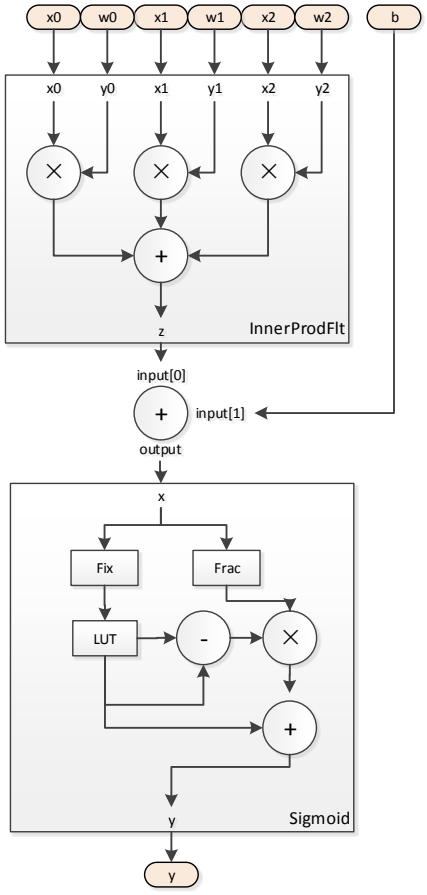

Figure 1 Flowgraph of the Perceptron from lines 37-55 in listing 1, implemented with its default property Inputs = 3.

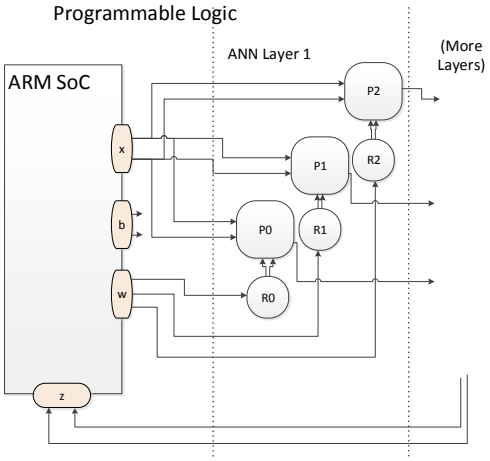

Figure 2 Implementation of an artificial neural network. Displayed is the implementation of a single layer in a cellnet ExampleAnn from lines 57 in listing 1 with an indication how it would extend to more layers. 'P' stands for Perceptron and 'R' stands for a reshaper. The connections of bias b are omitted for clarity.

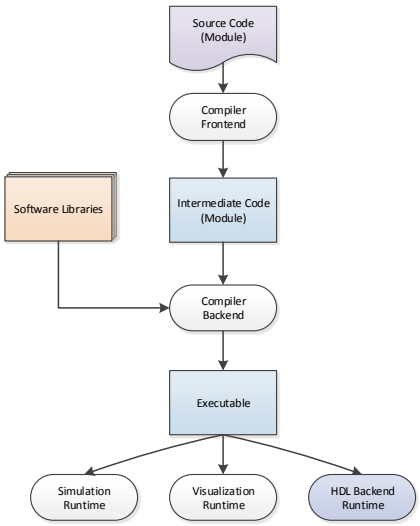

Figure 3 Compilation Process.

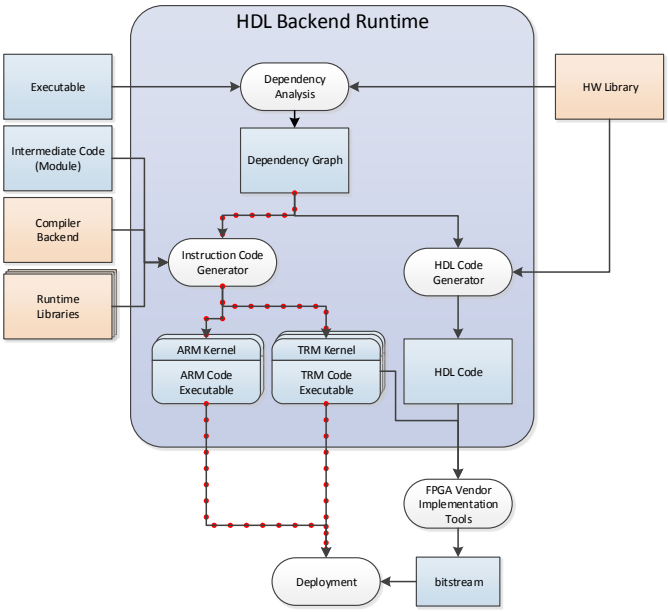

Figure 4 HDL Backend.

a perceptron, its body is executed, thereby creating cellnets and cells and returning only after all internal cells have been created and connected. Thereby, some of the internal ports are delegated to the outside, for example in line 47. The remaining code in the for-loop from line 70 establishes the connection of the delegated ports from Perceptron and the ports of the controller component. When the code from `ExampleAnn` returns, the cellnet is setup completely.

4 Hybrid Compilation

ActiveCells code is generally partitioned into modules. Modules can contain functions, variables, constants and types, including objects, cellnets and cells. Functionality in a module can be exported and consequently imported by others.

The unit of compilation is a module and there is separate compilation. We considered it important to adhere to this model and provide a system where a user can provide libraries to others without having to share source code. This has influenced some of the design decisions for the toolchain and the language. In the following we describe the compilation process as depicted with Figure 3.

The front end of the compiler generates an intermediate representation of the source code. This intermediate representation contains all executable parts of the source code, including cellnets and cells. From the intermediate representation the compiler generates (executable) object files.

Important in the context of this paper is that terminal cellnets can be executed completely on the development system. Communication actions such as connection, delegation, sending or receiving are mapped to special runtime calls and it depends on the chosen runtime what actually happens during execution on the development system.

For example, a runtime that spawns threads for cells during the (sequential) execution of cellnets can be used for *simulation*.

The HDL Backend runtime, displayed with Figure 4, is employed in order to *generate* a Network-on-Chip on an FPGA or on a hybrid FPGA-CPU system from a compiled cellnet. When the executable is cooperating with this runtime, it dynamically builds up a hierarchical network consisting of cellnets and cells. Driven by parameters passed to the executable it checks connectivity and consistency with the component- and architecture-description contained in the hardware library, cf. section 5. From this dependency graphs two separate paths are followed.

1. The instruction code generator infers from the dependency graph components that are deployed as processors. Examples of such processors are soft-cores like TRMs (“Tiny Register Machine” [37]) or prefabricated processors sitting within ARM SoC, fully integrated in Xilinx’ Zynq processors. The instruction code generator applies the respective compiler backend to the intermediate code, taking into account the processor properties stored in the dependency graph. The compiled code is then linked to a suitable runtime. Depending on the requirements and environment and type of the processor, this linked runtime can consist of a very tiny kernel providing only low-level memory management and instruction emulation support, or it provides more advanced operating system support such as device drivers, concurrency support, scheduling and virtual memory.

2. The HDL code generator transforms the dependency graph into HDL code, taking into account what kind of architecture was chosen from the command line parame-

ters. HDL code is passed on to an FPGA-vendor specific synthesis and implementation tool that executes hardware synthesis and generates a bitstream that can be deployed to hardware.

When both paths have been followed, the bitstream is deployed to an FPGA and the executables are integrated with the processors. This integration can be achieved by integrating the executable directly with the HDL code before hardware synthesis. Or it can be applied by patching the FPGA bitstream with the executable code with additional, vendor specific tools (such as Data2Mem offered by Xilinx). Or the code is provided to a boot-loader residing on one of the processor cores that distributes the executable code to all involved processor cores.

In any case there is a substantial time difference between the path that only generates the executables from software and the path that involved hardware synthesis and implementation. The fast software path, is highlighted in Figure 4 with red dots.

Why is this so interesting? It allows short implementation cycle times and therewith contributes to efficient debugging of complex solutions on FPGA. From an entrepreneurial viewpoint, this implies – together with the unified and comprehensible compute model and the portability and extensibility of the tool-chain – a short time-to-market.

A typical implementation of a complex SoC consists, according to our experience with various solutions for academia and industry, of the following implementation phases: (I) Initial design and setup, occurring once. (II) Minor changes, happening often and usually only affecting small portions of software code. (III) Major changes, happening sporadically, affecting the overall design and system topology.

With regards to the underlying tool-chain it is interesting to observe for the different phases the typical technical steps that have to be carried out in order to build the system.

(1) Compilation of the high-level program into data- and instruction stream for the programmable components (processors).

(2) Hardware synthesis and implementation

(3) Integration of the executables with their processors.

(4) Deployment to FPGA

When only minor changes have to be applied, usually the hardware synthesis and implementation step (2) can be omitted and the integration of executables (3) only have to be applied where necessary. By recording characteristics of the component graph that was used during compilation, the tool-chain automatically selects only the required actions necessary for each phase.

In Figure 5 we present implementation cycle times for the case studies described in more detail in Section 6. While case study B is an example particular chosen for this paper, case studies A and C provide real-world examples of systems that have been built in a commercial environment. In the figure, stream patching refers to the process of inserting compiled instruction streams into the memory files that are deployed to hardware. For case studies B and C this does not apply because the software part there ran on the ARM Cortex hardware of a Zynq 7000 FPGA. Therefore the deployment time is also higher for B and C – it contains a 5 seconds overhead for flashing the ARM component of the system. For case study A, the 'typical' patching size corresponds to toggling debugging output, a modification that affects a reasonable number of memory blocks. Patching one memory block of 2048 36-bit entries takes about one second.

Step	A	B	C
Software Compilation	4s	1 s	2 s
Hardware Implementation	1280 s	899 s	1280 s
Stream Patching (full)	136 s	-	-
Stream Patching (typical)	12 s	-	-
Deployment	11 s	16 s	16 s

Figure 5 Implementation cycle times for case studies A, B and C.

Note that for typical minor changes the overall compilation and deployment time is in the **order of several seconds** while for major changes and initial setup the compilation time is dominated by the hardware implementation time being in the **order of tens of minutes**.

5 Extensibility

The original implementation of [11] that was the starting point for the work described here was limited in extensibility of the hardware component and target platform libraries: Peripherals were designed as capabilities of soft-cores. This rendered different peripheral instances on the same or different core (e.g., multiple UARTs) indistinguishable from each other, Engines were thus treated differently from soft-core processors, precluding, hierarchical generic compositions. Support for different FPGA targets was realized by hardware-specific compiler modules, leading to an error prone redundancy and limited extensibility of the toolchain. Therefore, the the ActiveCells concept was extended to enable in a convenient and efficient way the integration of new FPGA hardware components into the HLL without compiler changes. To this end, generic interconnects and orthogonal integration of components and targets are described in the following paragraphs.

5.1 Communication Protocol

A fundamental property of the programming model described in Section 3 is the availability of configurable point-to-point interconnections between communicating agents (cells).

To improve the extensibility of the framework, a minimal communication protocol was defined that allows generic, pluggable connectivity of the involved components and provides high data throughput with low FPGA resource us-

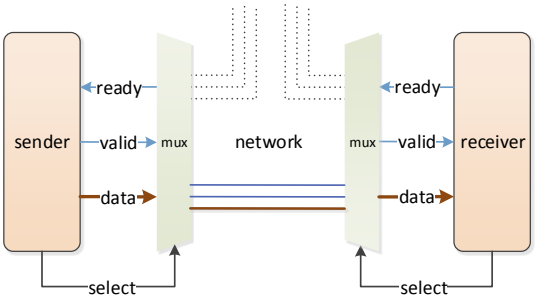

Figure 6 Signals for the Point-to-Point Interconnect.

age. The following signals are required for the point-to-point interconnects:

(a) Data: output data for data source and input data for data sink.

(b) Data available: source signals the availability of data to be sent.

(c) Recipient ready: sink signals the readiness to consume the data from the source.

The previous implementation in [11] that followed the principles of Kahn process networks did not support the ready signal (c) and, therefore, was prone to FIFO buffer overflows. To support multiplexing in a network of multiple sources and sinks, the following signals in addition to (a), (b) and (c) are required:

(d) Select sink: choose a target sink to which data and availability signals are directed and from which the Ready signal is read.

(e) Select source: choose a source from which data and availability signal are read and to which the Ready signal is directed.

Figure 6 outlines the required signals.
We found that AMBA AXI4-Stream Protocol defined in [1] satisfies the stated requirements. We therefore adopted a compatible subset of this protocol for our implementation. The AXI family of communication protocols is supported by the major FPGA vendors and is becoming the de-facto standard for interconnect interfaces between hardware components on FPGAs. Another emerging standard that was considered was the Wishbone computer bus [17]. Our decision to adopt AXI4-Stream protocol is mainly motivated by its striking simplicity.

5.2 Component Specification

Any cell defined in ActiveCells, irrespectively of its type (currently a processor or an engine), requires a description of the mapping from HLL to HDL and an algorithm for

```
type
  Gpo = cell{Engine,DataWidth=32,InitState="0H"}(input: port in);
```

```
module Gpo
#(
  parameter integer DW = 8,
  parameter InitState = {DW{1'b0}}
)
(
  input aclk,
  input aresetn,

  input [DW-1:0] inp_tdata,
  input inp_tvalid,
  output inp_tready,

  output [DW-1:0] gpo
);
```

Figure 7 ActiveCells HLL interface (top) and Verilog HDL interface (bottom) of component defined in Listing 2

generating HDL code. We decided to implement them in HLL, similar to the components, yielding a high degree of flexibility and consistency. An example of a specification is presented in Listing 2, describing a hardware component implementing a General Purpose digital Output (GPO). A component specification defines the following:

- **component identification**: map HLL cell name to HDL module name, here `Gpo` $\mapsto$ `AxisGpo`,
- supported **Programmable Logic Devices** (PLD),
- component HLL **properties** and their HDL equivalents, e.g. Verilog HDL parameters,
- component **connectivity interfaces**: mapping of HLL cell ports to HDL ports. Figure 7 presents HLL and HDL interfaces of the Gpo component from the example above,
- algorithm for **HDL code generation** including a set of activities determined by the component parameters and the HDL source dependencies. This can range from simple inclusion of existing HDL-Code up to full-fledged HDL-Code emission.
- **registration** in the ActiveCells hardware library.

Once the component has been specified it becomes immediately available to the ActiveCells tool-chain for automatic HDL code generation.

5.3 Target device specification

For compilation of an ActiveCells architecture, a description of the concrete target device is required. Listing 3 presents an example of a target specification for a Xilinx ARTIX-7-based evaluation board from Digilent Inc. A target specification defines

- **FPGA part**: part name, speed grade, package type etc.

```
1  module Gpo;
2  import Hdl := AcHdlBackend;
3  var c: Hdl.Engine;
4  begin
5    (* Define component identification *)
6    new(c,"Gpo","AxisGpo");
7
8    (* Define supported PLD devices *)
9    c.SupportedOn("*"); (* portable *)
10
11   (* Define parameterization *)
12   c.NewProperty("DataWidth","DW",Hdl.NewInteger(32),
13                 Hdl.IntegerPropertyRangeCheck(1,Hdl.MaxInteger));
14   c.NewProperty("InitState"," InitState ",
15                 Hdl.NewBinaryValue("0H"),nil);
16
17   (* Define port interface *)
18   c.SetMainClockInput("aclk");
19   c.SetMainResetInput("aresetn",Hdl.ActiveLow);
20   c.NewAxisPort("input","inp",Hdl.In,32);
21   c.NewExternalHdlPort("gpo","gpo",Hdl.Out,32);
22
23   (* Define influence of parameters on HDL implementation *)
24   c.AddPostParamSetter(Hdl.SetPortWidthFromProperty("inp","DW"));
25   c.AddPostParamSetter(Hdl.SetPortWidthFromProperty("gpo","DW"));
26
27   (* Define HDL code *)
28   c.NewDependency("Gpo.v");
29
30   (* Register component in the library *)
31   Hdl.hwLibrary.AddComponent(c);
32 end;
```

Listing 2 Specification of an ActiveCells hardware component

```
1  module Basys3Board;
2  import
3    Hdl := AcHdlBackend;
4  var
5    t: Hdl.TargetDevice;
6    ioSetup: Hdl.IoSetup;
7  begin
8    new(t,"Basys3Board",
9        Hdl.NewXilinxPldPart("XC7A35T-1CPG236"));
10
11   (* Define system clock signal *)
12   t.NewExternalClock(
13     Hdl.NewIoPin(Hdl.In,"W5","LVCMOS33"), 100.0E6,50.0,0.0);
14   t.SetSystemClock(
15     t.GetClockByName("ExternalClock0"),1.0,1.0);
16
17   (* Define system reset signal *)
18   t.SetSystemReset(
19     Hdl.NewIoPin(Hdl.In,"U18","LVCMOS33"),Hdl.ActiveLow);
20
21   (* Define IO mapping for a Gpo component instance *)
22   new(ioSetup,"Gpo_0");
23   ioSetup.NewIoPort("gpo",Hdl.Out,"U16,E19,U19,V19","LVCMOS33");
24   t.AddIoSetup(ioSetup);
25
26   (* Register target in the library *)
27   Hdl.hwLibrary.AddTarget(t);
28 end Basys3Board.
```

Listing 3 Specification of an ActiveCells target platform

- **system signals**: system clock and system reset signals with their parameters, and IO pin specification; possibly other clock signals available on the target device,
- mapping of external ports of a component to **FPGA IO pins**, including signal direction, IO standards, etc. Observe the decoupled definition of external (terminal) ports in the component specification in Listing 2, line 21, and the corresponding mapping to FPGA IO pins in Listing 3, line 23.
- other target-specific constraints

The generic communication protocol, component specification, and target specification together with the program in HLL form the necessary basis for fully automatic generation of HDL projects and their implementation using vendor-specific hardware design tools as a transparent backend.

5.4 Hierarchical HDL Code Generation

In [11] the dependency tree extracted from the ActiveCells code was flattened during HDL code generation, aiming at a simplification of the process. However, this approach can lead to an explosion of automatically generated HDL source code: this problem occurs in particular when a piece of hierarchical ActiveCells code is replicated (e.g. multiple instances of the same cellnet in a multi-channel design). We have addressed the problem now by mirroring the hierarchical structure of the ActiveCells code in HDL code generation.

Technically, the compilation process is preceded by a short browsing phase, where the compiler traverses the directory of all implemented components and records all mappings such that during compilation the right kind of components, pins etc. can be assigned. Using a compilation flag, the compiler is informed about the specific target architecture.

6 Case Studies

In this section we provide three case studies that demonstrate the versatility of the toolchain and to explore the performance of real-world applications.

6.1 Case Study A - Hemodynamic Monitor

This is a real-world computational medicine example implemented with parallel soft-core components on an FPGA. The system replaced a predecessor on a single standard processor running on several threads. The whole system could be implemented with our approach in a **radically simplified form** because task management was rendered superfluous.

The system is illustrated in Figure 9.

The system implements a medical monitor with hardware in the loop, for sensing, motor control and real-time computation of hemodynamic parameters such as *cardiac output*, the amount of blood volume pumped by the heart per time unit. The realtime challenge for this system is continuous control of the sensor position, driven by the outputs

System	Architecture	Resources	Clock Rate	Performance	Data Bandwidth
A	Spartan 6 XC6SLX75	28% Slice LUTs, 4% Slice Registers 80% BRAMs, 24% DSPs	58 MHz	- at ∼2W	1.25 MBit/s 23 kB/s (out)
B	Zynq 7000 XC7Z020	11% Slice LUTs, 6% Slice Registers 7% BRAMs, 15% DSPs 1 PS 7 (ARM Cortex A9)	118 MHz	8.3 GFlops at ∼5W	236 MWords/s (in) 118 MWords/s (out)
C	Zynq 7000 XC7Z020	17% Slice LUTS, 8% Slice Registers 22% BRAMs, 31% DSPs 1 PS 7 (ARM Cortex A9)	50 MHz	4.3 GFlops at ∼5W	50 MWords/s (in) 50 MWords/s (out)
D	Zynq 7000 XC7Z010	83% Slice LUTS, 67% Slice Registers 13.33% BRAMs, 21% DSPs 1 PS 7 (ARM Cortex A9)	147 MHz	4.9 GFlops at 2.1W	9.6 GBits/s (in/out)

Figure 8 Resource usage, clock rates and performance related numbers for case studies A, B and C.

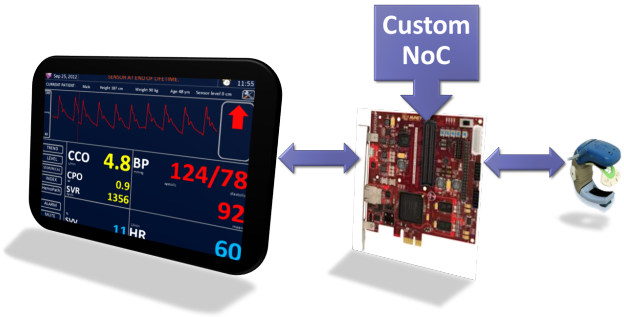

Figure 9 Illustration of the hemodynamic monitor system: Spartan-6 FPGA (middle) connected to medical monitor (left) and to blood pressure sensor (right).

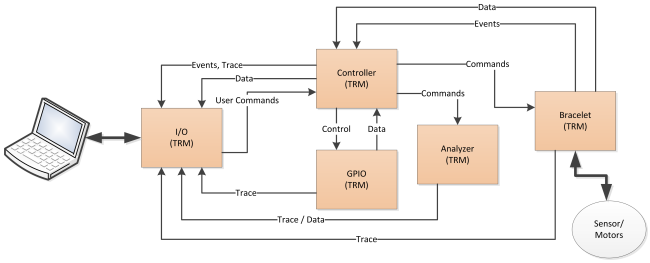

Figure 10 The Network-on-Chip of the hemodynamic monitor. This network consists of very small processors (Tiny Register Machines - TRMs) with some of them equipped with a floating point unit.

of non-trivial algorithms that determine characteristics of the continuous blood pressure signal acquired at a rate of 162 Hz. Sensor readout and sensor position control are performed at a rate of 4 kHz.
Figure 10 provides a view on the Network-on-Chip that is deployed on the FPGA for this system.

6.2 Case Study B - Dispersion Compensation

With this case study, we present an implementation of a part of a dispersion compensation block (cf. [38]) that is used in Optical Coherence Tomography (OCT) imaging systems (described, for example, in [3]).
This system which focuses on a high throughput application in a FPGA-CPU hybrid SoC consists of set of hardware components interconnected with a Network-on-Chip. Hardware components communicate with each other via AXI4-Stream ports, as described in 5.1. This applies also to the controller, which is implemented on a built-in ARM Core on a Xilinx Zynq-7000 FPGA platform. The ARM component on this system is equipped with a set of AXI4 memory mapped interfaces. In order to provide AXI4-Stream communication between the controller and the other components, we designed a lightweight converter that maps AXI4 memory mapped interfaces to the AXI4-Stream interfaces. In order to seamlessly integrate the ARM core into our framework, we defined in our hardware library that the ARM component is equipped with such converters. The input of image data to the controller from an external high performance camera is implemented via GBit Ethernet, with a second GBit Ethernet for image data output. The received image data is transmitted to the programming logic using the DMA controller of the ARM CPU. For this particular example, we use general purpose AXI4 interfaces with a maximal data throughput of 4.8 GBit/s each, independently for both directions. The complex-valued signal is processed sample by sample at every clock cycle. First, complex multiplication of the input signal with weight factors stored in cyclic buffers is performed. This complex valued weighted result is fed to an FFT component. The squared absolute value from the FFT is computed. Note that each component is configured from the controller, e.g. determining the right-shift value of fixed-point multiplication. Figure 8 lists the resources used and shows performance numbers for this system. We were able to deploy a system that consisted of four instances of such dispersion compensation block that were all connected to AXI4-Stream ports of the ARM component, reaching an overall **computational performance of more than 32 GFlops** at a power consumption of about

5W.

6.3 Case Study C - Optical Coherence Tomography

Using the proposed programming model and tool-chain, a proprietary OCT system was implemented, which is now in real use in a medical imaging device, also containing as a subsystem the dispersion compensation block presented before as case study in Section 6.2. In addition, it implements a high-quality image reconstruction subsystem, that poses a serious challenge for being efficiently pipelined. Therefore we observe significant reduction of the achievable clock rate, as visible in Figure 8. Still, the achieved performance is substantial, and the use of multiple clock domains could further improve throughput.

6.4 Case Study D - Perceptron

A single perceptron (cf. Figure 1) with 15 inputs was implemented on a Zynq XC7Z010-1CLG400, the smallest FPGA in the Xilinx Zynq family. The frequency of this design implemented using single precision floating point arithmetics was 147 MHz, reaching a computational performance of 4.9 GFlops. The resource usage is subsumed in Figure 8. We measured a power consumption of 2.125W while streaming to/from PL. The measurements showed a power consumption of 2W when the FPGA was not programmed at all. The design throughput is currently bound to the maximal bandwidth of master AXI4 interfaces, i.e. 9.6 GBit/s in both directions. The bandwidth can be increased with the use of four High-Performance AXI4 slave interfaces, which according to the specifications allow throughput of 38.4 GBit/s in both directions.

7 Conclusion and Future Work

We presented an approach to high-performance system design and implementation on FPGAs and hybrid FPGA-CPU systems using a high level, programming model with a unified development flow.

The high readability of the language, the unified compilation for hardware and software and the possibility for system simulation facilitate debugging and prototyping. As a side effect, the work-flow leads to short development cycle times and consequently to rapid prototyping and short time-to-market as proven in the development of currently marketed products. An important innovative aspect of the tool-chain described in this paper is the achieved extensibility, maintainability and portability. Complementary case studies showed that this approach is advantageous for rapid construction of flexible, runtime configurable high-performance hybrid SoCs.

Currently, the process of development of IP blocks requires an experienced FPGA engineer and the use of low level hardware description languages, e.g. Verilog or VHDL. Initial work has been done in extending the language with High-Level Synthesis capabilities in order to simplify the extension of the hardware library base of the framework. We have concrete ideas of how new components can be constructed from existing ones by an automated latency analysis for component pipelines. To further advance in heterogenous high performance computing, the framework will be extended to support multi-FPGA, multi-core hybrid platforms for addressing further interesting problems, in particular in high-performance medical computing.

8 Literature

[1] *AMBA 4 AXI4-Stream Protocol, Specification Version 1.0*. ARM, 2010.

[2] Gul Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA, USA, 1986.

[3] Murtaza Ali and Renuka Parlapalli. *Signal Processing Overview of Optical Coherence Tomography Systems for Medical Imaging*. Texas Instruments Incorporated, 2010.

[4] Josh Auerbach, Dave F. Bacon, Perry Cheng, Steve Fink, and Rodric Rabbah. The shape of things to run: Compiling complex stream graphs to reconfigurable hardware in lime. In *Proceedings of the 27th European Conference on Object-Oriented Programming*, ECOOP'13, pages 679–706, Berlin, Heidelberg, 2013. Springer-Verlag.

[5] Joshua Auerbach, David F. Bacon, Perry Cheng, and Rodric Rabbah. Lime: a java-compatible and synthesizable language for heterogeneous architectures. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '10, pages 89–108, New York, NY, USA, 2010. ACM.

[6] J. Bachrach, Huy Vo, B. Richards, Yunsup Lee, A. Waterman, R. Avizienis, J. Wawrzynek, and K. Asanovic. Chisel: Constructing hardware in a scala embedded language. In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, pages 1212–1221, June 2012.

[7] David Bacon, Rodric Rabbah, and Sunil Shukla. Fpga programming for the masses. *Queue*, 11(2):40:40–40:52, February 2013.

[8] J. Cong, Bin Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Zhiru Zhang. High-level synthesis for FPGAs: From prototyping to deployment. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 30(4):473–491, April 2011.

[9] T.S. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, and D.P. Singh. From opencl to high-performance hardware on FPGAs. In *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, pages 531–534, Aug.

[10] J. B. Dennis. First version of a data flow procedure language. In *Programming Symposium, Proceedings Colloque sur la Programmation*, pages 362–376, London, UK, 1974. Springer-Verlag.

[11] Felix Friedrich, Ling Liu, and Jürg Gutknecht. Active cells: A computing model for rapid construction of on-chip multi-core systems. In *ACIS-ICIS'12*, pages 463–469, 2012.

[12] Robert Griesemer, Rob Pike, and Ken Thompson. The go programming language, http://golang.org. *Website, accessed at 19.1.2016*, 2016.

[13] Jürg Gutknecht. *SOFSEM 2011: Theory and Practice of Computer Science: 37th Conference on Current Trends in Theory and Practice of Computer Science, Nový Smokovec, Slovakia, January 22-28, 2011. Proceedings*, chapter A Structured Codesign Approach to Many-Core Architectures for Embedded Systems, pages 15–25. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[14] Per Brinch Hansen. Joyce - a programming language for distributed systems. *Softw., Pract. Exper.*, 17(1):29–50, 1987.

[15] Per Brinch Hansen. The origin of concurrent programming. chapter SuperPascal: A Publication Language for Parallel Scientific Computing, pages 495–524. Springer-Verlag New York, Inc., New York, NY, USA, 2002.

[16] T. Harriss, R. Walke, B. Kienhuis, and E. Deprettere. Compilation from matlab to process networks realised in fpga. In *Signals, Systems and Computers, 2001. Conference Record of the Thirty-Fifth Asilomar Conference on*, volume 1, pages 458–462 vol.1, Nov 2001.

[17] Richard Herveille et al. Wishbone system-on-chip (soc) interconnection architecture for portable ip cores. *OpenCores Organization, September*, 2002.

[18] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd international joint conference on Artificial intelligence*, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[19] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21:666–677, August 1978.

[20] Amir Hormati, Manjunath Kudlur, Scott Mahlke, David Bacon, and Rodric Rabbah. Optimus: Efficient realization of streaming applications on fpgas. In *Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '08, pages 41–50, New York, NY, USA, 2008. ACM.

[21] Shan Shan Huang, Amir Hormati, David F. Bacon, and Rodric Rabbah. Liquid metal: Object-oriented programming across the hardware/software boundary. In *Proceedings of the 22Nd European Conference on Object-Oriented Programming*, ECOOP '08, pages 76–103, Berlin, Heidelberg, 2008. Springer-Verlag.

[22] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. Advances in dataflow programming languages. *ACM Comput. Surv.*, 36:1–34, March 2004.

[23] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information processing*, pages 471–475, Stockholm, Sweden, Aug 1974. North Holland, Amsterdam.

[24] O. Mencer. Asc: a stream compiler for computing with fpgas. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 25(9):1603–1617, Sept 2006.

[25] Oskar Mencer. Maximum performance computing for exascale applications. In *ICSAMOS*, 2012.

[26] Pieter J. Muller. *The Active Object System Design and Multiprocessor Implementation*. PhD thesis, ETH Zürich, 2002.

[27] A.H. Namin, K. Leboeuf, Huapeng Wu, and M. Ahmadi. Artificial neural networks activation function hdl coder. In *Electro/Information Technology, 2009. eit '09. IEEE International Conference on*, pages 389–392, June 2009.

[28] Florian Negele. *Combining Lock-Free Programming with Cooperative Multitasking for a Portable Multiprocessor Runtime System*. PhD thesis, Diss., Eidgenössische Technische Hochschule ETH Zürich, Nr. 22298, Zürich, 2014.

[29] Florian Negele, Felix Friedrich, Suwon Oh, and Bernhard Egger. On the design and implementation of an efficient lock-free scheduler. In *19th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, JSSPP 19, 2015.

[30] P.R. Panda. Systemc - a modeling platform supporting multiple design abstractions. In *System Synthesis, 2001. Proceedings. The 14th International Symposium on*, pages 75–80.

[31] A. W. Roscoe and C. A. R. Hoare. The laws of occam programming. *Theor. Comput. Sci.*, 60(2):177–229, September 1988.

[32] D. Sheldon, R. Kumar, R. Lysecky, F. Vahid, and D. Tullsen. Application-specific customization of parameterized fpga soft-core processors. In *Computer-Aided Design, 2006. ICCAD '06. IEEE/ACM International Conference on*, pages 261–268, Nov 2006.

[33] S. Singh and D. Greaves. Kiwi: Synthesis of fpga circuits from parallel programs. In *Field-Programmable Custom Computing Machines, 2008. FCCM '08. 16th International Symposium on*, pages 3–12, April.

[34] S. Skalicky, T. Kwolek, S. Lopez, and M. Lukowiak. Enabling fpga support in matlab based heterogeneous systems. In *ReConFigurable Computing and FPGAs (ReConFig), 2014 International Conference on*, pages 1–6, Dec 2014.

[35] John E. Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *IEEE Des. Test*, 12(3):66–73, May 2010.

[36] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. Streamit: A language for streaming applications. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages

179–196, London, UK, UK, 2002. Springer-Verlag.

[37] Niklaus Wirth. The Tiny Register Machine (TRM). Technical Report 643, ETH Zürich, Computer Systems Institute, 10 2009.

[38] Maciej Wojtkowski, Vivek Srinivasan, Tony Ko, James Fujimoto, Andrzej Kowalczyk, and Jay Duker. Ultrahigh-resolution, high-speed, fourier domain optical coherence tomography and methods for dispersion compensation. *Opt. Express*, 12(11):2404–2422, May 2004.

[39] Zhiru Zhang, Yiping Fan, Wei Jiang, Guoling Han, Changqi Yang, and Jason Cong. Autopilot: A platform-based esl synthesis system. *High-Level Synthesis: From Algorithm to Digital Circuit*, pages 99–112, 2008.