Active Cells – A Computing Model for Rapid Construction of On-Chip Multi-Core Systems

Felix Friedrich, Ling Liu, Jürg Gutknecht
ETH Zürich
Computer Systems Institute
Zürich, Switzerland
{*felix.friedrich, ling.liu, gutknecht*}*@inf.ethz.ch*

***Abstract*—We present a novel computing model that allows to conveniently construct multi-core systems with different computer architectures, ranging from homogeneous many-core architectures to networks of heterogeneous general purpose processor cores or signal processing engines.**

A hardware library implemented on Field Programmable Gate Arrays (FPGAs) and a compiler provide a platform for prototyping and constructing distributed systems on a chip. A number of case studies have been carried out to prove the concept conveyed by the computing model.

***Keywords*-Parallel programming; Concurrent computing; Distributed computing; Multicore processing; FPGA design; System-level design; Software-hardware co-design;**

I. Introduction

We think that it should be possible for a developer to design application specific hardware architecture on an FPGA in a simple and convenient way. However, the huge design overhead introduced by separate software hardware design processes, tools and programming models today makes this difficult. Developers usually purchase a commercial, say multi-core, processor and develop their systems, also with a high dependence on the operating system underneath. Then they are confronted with bottlenecks in communicating over shared memory, high process synchronization overheads and all typical complexities in programming a multi-threaded operating system. In our vision, the implied pecularities of a specific processor architecture, processor sharing, interrupts, global shared memory, paging etc. should only be involved in hardware if such features are strictly necessary or bring vital performance or energy benefits.

A programming model and a platform that allow on-chip system developers to automatically build the target hardware architecture from the structure of the target application is definitely on demand and helpful. Here we present such a model. However, it cannot be the goal of our work to solve all problems of the multi-core computing world. We concentrate on the class of data-driven streaming applications for embedded computing. [1]

[1]We take advantage of the typical problem structure by featuring task parallelism (simultaneous execution), stream parallelism (pipelined execution) and data parallelism (vector computing / loop-level parallelism) in the model and our tools.

In our previous work, we started with research on an implementation of simple processor cores ([1]) on FPGA together with a network on chip that was designed to support distributed applications ([2], [3], [4]). The idea to set up connections between cores if and only if it is necessary from the application point of view brought us to the point where real *application-aware hardware* could be built with optimized performance, energy efficiency and use of hardware resources ([5], [6]).

This work has been brought further and resulted in the computing model *Active Cells*, which takes the best of two worlds: flexibility of software together with the efficiency of programmable hardware. Paired with the simplicity from our programming model. The model includes a tool set that allows system developers to prototype or construct their systems on programmable hardware. By the provision of a simple but powerful, consistent, self-contained programming language, programming environment and all tools necessary to compile the high-level language down to programmable hardware, the computing model can abstract away all details and difficulties from the programmer. Programmers are not required to have deep knowledge in hardware design. The following figure summarizes this idea.

System design as high level program code	⟹	**Electronic Circuits FPGA, ASIC**
Active Cells Program	Compiler Synthesizer Simulator	Programmable Hardware Description

The rest of this paper is organized as follows: Section II introduces the FPGA-based hardware components used to emulate hardware architecture. Section III introduces and illustrates the programming model. Section IV describes the implementation. Section V presents the case study results of applying Active Cells computing model to three different real-time on-chip systems. Section VI concludes the paper.

II. The Hardware Library

The hardware library implemented on FPGAs provides different computing elements, ranging from general purpose RISC processors and vector processors to digital signal processing engines. The main components are:

- A tiny RISC machine (TRM), a two address processor with 18 bit instruction format and 32 bit data path. Each TRM contains an arithmetic-logic unit (ALU), a multiplier, a barrel shifter and 8 working registers. The 2-stage pipelined implementation of a TRM runs at 116MHz, and takes 2% LUTs of the Virtex-5XC5VLX50T FPGA. The multiplication in the TRM takes 5 clock cycles.
- A vector processor VTRM has been developed to support the data parallelism required by the computation processes.
- Configurable FIFOs. FIFOs are implemented with LUTs or BRAMs depending on the depth of the FIFO.
- A DDR2 interface.
- I/O controllers. A compact flash (CF) controller, a LCD controller and a UART controller have been implemented for Xilinx ML505 board. They all run at 116MHz. A VGA controller and a DVI controller have also been implemented. They run at pixel clock.
- Special processing elements such as
  - Configurable multiplier and accumulator (MAC). A MAC is the key component of signal processing units.
  - Configurable two-dimensional convolution engine (2DConvolver) commonly used in signal processing, especially image processing systems.

To bridge the gap between cycle accurate hardware design and function centric software design, a set of hardware interfaces have been implemented in a hardware description language to allow compilation tools to address the specific hardware. The following items provide some examples of the hardware interfaces defined in Verilog code. Notations following the symbol "#" describe parameters that can be configured from an application, for example the *instruction memory block size (IMB)*.

- TRM interface.
  ```
  #(IMB, DMB) TRM (input clk, rst, irq0,
  irq1, input[31:0] inbus, output[5:0]
  ioadr, output iowr, iord, output[31:0]
  outbus)
  ```
- Channel interface.
  ```
  #(Width, Depth) ParChannel (input clk,
  rst, input[Width-1:0] inData, input
  wreq, rdreq, output[Width-1:0] outData,
  output[31:0] status)
  ```

III. The Active Cells Programming Model

The programming model Active Cells provides a way to describe, in an abstract way, a programmable system composed of computing elements that exchange information over message channels. The programming model has primarily been designed to support the simple definition and programming of multi-core systems on programmable hardware, but it has also been envisaged to be applicable to custom, general purpose multi-core hardware.

Experiences with building multi-core systems on FPGAs have led to the further requirement that the processor cores do not immediately share memory and thus form a *distributed system of isolated processes, equipped with local memory and communicating via message passing*.

The herein described programming model features so called *Active Cells*, objects with a private state space and integrated control thread(s). Active Cells can be connected via channels to enable communication. Moreover, Active Cells can be aggregated in so called *Cell Nets*, networks of communicating cells. Cell nets, i.e. aggregated cells, can also be connected with cells or other cell nets.

In order to support the concept of Active Cells, the following features have to be added to a general purpose programming language:

- Cells: a cell provides the scope of a process that runs in an isolated environment. Cells are defined as types in a type declaration section.
- Ports: cells can exchange data via unidirectional channels. Cells can be connected using channels between input and output ports of cells. Cells can send or receive data over ports using built-in primitives *send* and *receive*. Sending and receiving is buffered by default. Sending is non-blocking. Receiving exists in a blocking and an unblocking version, decided by the actual parameters.
- Connections: ports are connected using unidirectional channels using the *connect* statement. Channels are not made explicit in the language.
- Cell nets: a cell net defines a directed graph over cells or cell nets. Cell nets can provide ports that are delegated to ports of the contained components using the primitive *delegate*.

The programming model that we propose has been inspired by many works on data flow languages ([7], [8]), Kahn Process Networks ([9]), seminal works on parallel computing (such as [10]) and the actor model ([11], [12]). The formulation as a compositional framework follows very much [13].

Although the concept we present here is universal and can be applied to other programming language, we exemplify our approach with the language that we use for our development, Active Math Oberon. Active Oberon is a type safe, object oriented programming language in the tradition of Pascal and

Modula and provides a concept for concurrent execution as part of the language ([14], [15]). We have amended Active Oberon with a Matlab-like syntax for mathematical programming ([16], [17]). It can be used to describe mathematical algorithms in a high-level notation and make immediate use of vector capabilities of underlying hardware, a concept that obviously fits very well to the scope of this work.

A. Cells

A cell provides the scope and environment for a running, isolated process. Cells do not immediately share memory but can only communicate via channels. Cells are defined as types with a scope that can contain variables, procedures and a body. The body of a cell provides the code for the primary activity associated with the cell. A number of input and output ports can be defined as parameters of a cell. Such ports define the interface of the cell and nothing else is visible to both the interior scope of the cell and the connected cells. The direction of the port must be defined to be *in* or *out*.

Listing 1 illustrates this with an implementation of a cell that receives two incoming values and that returns the result of some subsequent operation over an outgoing stream. Note that receiving is blocking in the displayed form. A non-blocking receive exists and can be expressed using the ternary form *receive(port, value, result)*.

```
type F = cell (in1, in2: port in; res: port out);
  var i,j: integer;

  procedure SomeOp (x,y: integer): integer;
  begin ... return ...
  end SomeOp;

begin
  loop
    (* blocking receive *)
    receive(in1, i); receive(in2, j);
    (* non-blocking send *)
    send(res, SomeOp(i,j))
  end
end F;
```

Listing 1. A communicating cell.

Capabilities: Applied to the code of Listing 1, our compiler would by default generate a programmable processor core on a chip with defaults in terms of supported instruction set, memory sizes, port widths etc. It is possible to override default values and to configure the component to implement a different instruction set, to incorporate additional features such as a vector processing unit or connections to devices etc. This is accomplished by the specification of capabilities in the declaration of a cell. The capabilities of a cell can influence what the synthesized hardware components looks like and what kind of code is generated by the compiler. [2]

Listing 2 contains an example of a cell that is configured to contain a vector processing unit, a data memory of 2048 words and a connection to a DDR2 memory controller. Moreover, the incoming port is configured to be 64-bit wide.

```
type Filter = cell { Vector, Data(2048), DDR2 }
                (data: port in (64); res: port out);
  var t: real; k1, ..., x : array [3] of real;(* vectors *)
begin
  ...
  while t <= tmax do
    (* vector operations *)
    k1 := f(t, x); k2 := f(t + dt/2, x + dt/2 * k1); ...
    x1 := x + dt/3*(1/2*k1 + k2 + k3 + 1/2*k4); ...
  end
end Filter;
```

Listing 2. A cell with capabilities

Engines: As indicated before, cells usually represent programmable processor cores providing control unit, arithmetic unit and registers. For some tasks, the usage of pure hardware implemented engines is a better choice. For example, very simple components such as moving average, adder or threshold filter would waste resources in terms of space and power consumption on an FPGA if they were implemented as a fully-fledged processor. Other components, such as more complex image filters, may provide a higher throughput, lower latencies and thus a better performance if they are implemented directly on hardware, cf. the description of special processing elements in Section II.

Therefore, we provided a way to designate a cell to be implemented in hardware. We call such a cell an Engine. Listing 3 provides an example how such an Engine is defined in Active Cells. The availability of a hardware implementation of the Engine is checked by the compiler.

```
type Convolver2D = cell {Engine}
        (raw: port in; filtered : port out);
end Convolver2D;
```

Listing 3. An Engine cell made from hardware

B. Cell Nets

Single cells can be equipped with capabilities to provide a connection to the outside world, for example over attached controllers. However, the interconnection of cells must be made explicit with a definition in an outer scope. Setting up connections within the scope of a cell would, in general, require dynamic composition. And dynamically allocating hardware resources is a time-consuming procedure in this context and can thus hardly meet systems' real-time requirements.

Several alternatives were considered, such as the definition of the graph of cells in an external XML file, a graphical composition framework etc. The following requirements have to be met

(1) It must be easy to construct complex, parameterizable graphs of communicating processes.
(2) The programming model should be applicable to the development on programmable hardware and to building solutions on conventional multi-core hardware. It

[2] More details on the compilation process are provided in Section IV.

should thus cover the static construction of hardware cores and the dynamic construction of threads.

(3) The programming model should be easy to learn and to teach, consistent and as simple as possible.

To account for (1) and (2), a programming language comprising control flow constructs was envisioned for the construction of the graph. To account for (3), we decided that for the cell net composition the same language should be used as for programming the cells. With the consequence, that the compiler had to be able to interpret certain parts of the code during compilation.

We follow a three phase composition model: instantiate, connect, initialize. Like cells, cell nets are defined as types with a body where the configuration of a network of cells takes place. A *terminal* cell net, i.e. a cell net that does not provide any ports can form a compilation and deployment unit.

1) Wiring Cells in a Cell Net: The *new* statement is used on a variable of cell (or cell net) type to instantiate a cell (or cell net). If there is a constructor available in the cell, it is possible to pass initialization parameters. The *connect* statement can be used to connect an outgoing port of one component to an ingoing port of another. The depths of two connected ports, i.e. the sizes of the associated FIFO, can also be specified as a third parameter of *connect*.

Listing 4 contains an example of a terminal cell net *A* comprising two cells, one user interface cell *ifc* of type *UI* providing communication to the outside world over RS232, and a communicating cell *f* of type *F*. The implementation of a serial connection is provided in the imported module RS232.

```
cellnet A; (*terminal*)
import RS232;
type
  F = cell (in1, in2: port in; res: port out);
  UI = cell {RS232} (out1, out2: port out; res: port in);
var
  ifc: UI; f: F;
begin
  (* creation *)
  new(ifc); new(f);
  (* wiring *)
  connect(ifc.out1, f.in1);
  connect(ifc.out2, f.in2);
  connect(f.res, ifc.res)
end A.
```

Listing 4. A terminal cellnet; implementation of F and UI omitted

A compilation of the displayed cell net *A* results in the generation of code for the body of the cell type *UI*, code for the body of *F* and a network description that contains the wiring defined in the body of the cell net. The network description contains references to linked code being ready for execution and can be used to deploy the example, either to hardware or a simulator.

2) Hierarchic Composition: Cell nets constitute the wiring of cells (and cell nets). To make a connection of a whole cell net to other cells (or cell nets) possible, cell nets

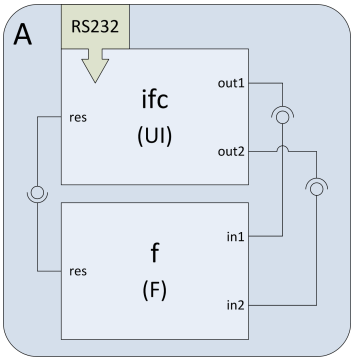

Figure 1. Topology of the terminal cellnet as defined in Listing 4. We write types of a cell in parentheses to distinguish from the instance names. Names are relative to enclosed scopes.

can also provide ports. Ports of a cell net can be delegated to ports of contained components using the *delegate* primitive. Naturally, the scope of a cell net can also contain the definition of locally used cell types and (sub-) cell nets. Instances of cells or cell nets are represented as variables in the cell net. Thus, instances of cell nets can form networks of instances of cells and instances of cell nets.

Listing 5 contains an example of how components would be stored to form a reusable library. The contained reusable cell net *ScalarProduct* wires two multipliers and an adder (engines) such that they form a scalar product.

```
namespace MathLib;

  type MovingAverage* =cell (in: port in; res: port out);
    procedure &Init(length: integer); (* constructor *)
  ... end MovingAverage;

  type Adder* =cell {engine} (in1,in2: port in;
                    res: port out);
  ... end Adder;

  type Multiplier* =cell {engine} (in1,in2: port in;
                         res: port out);
  ... end Multiplier;


  type ScalarProduct* =cellnet(vX,vY,wX,wY: port in;
                               res: port out)
  var
    adder: Adder;
    mul1,mul2: Multiplier;
  begin
    new(mul1); new(mul2); new(adder);
    delegate(vX, mul1.in1); delegate(wX, mul1.in2);
    delegate(vY, mul2.in1); delegate(wY, mul2.in2);
    delegate(res, adder.res);
    connect(mul1.res, adder.in1);
    connect(mul2.res, adder.in2);
  end ScalarProduct;

end MathLib.
```

Listing 5. A Library consisting of cells and cell net

IV. Implementation

When an Active Cells program is compiled, parts of it are necessarily compiled "to hardware". This means they are

compiled to a hardware specification that can be understood by hardware synthesis tools to generate components on programmable hardware. Other parts have to be compiled to code that is executable on the synthesized processors amongst the hardware components. The mapping from software to hardware plus code is described in this section and can be summarized as follows:

Cell Net		FPGA
Cell		Engine or TRM processor Instruction & Data Memory
Communication Channel	$\Longrightarrow$	FIFO buffer
I/O		I/O controllers

A. Hybrid Compilation

A cell stands for a programmable processor (if it is not tagged as engine). Thus the code of the body of a cell is compiled to executable code on the processor. The way the code is compiled can depend on the features of the cell. For example, if a cell is flagged to contain an FPU unit, the generated hardware will contain an FPU unit and floating point operations are compiled to FPU instructions. The front-end of our compiler generates intermediate code that is passed to a back-end for the particular architecture. This usage of intermediate code for the generated code makes separate compilation at all possible in this context.

A cell net stands for a network of processors and therefore forms the unit of what is compiled to hardware. We are not considering the generation of programmable hardware at runtime but merely have to rely on synthesis tools that are available on the development machine. Therefore the bodies of networks have to be interpreted during compilation in order to generate a hardware specification that can be understood by synthesis tools.

The automated process for mapping an Active Cells program onto an FPGA is outlined in Figure 2.

B. Runtime Library

The sending and receiving over channels requires a considerable support by a runtime library. During compilation, both for the hardware generation and code generation part, each port of a cell is associated with a number that is used for addressing the port. In fact, this number stands for the memory-mapped addresses of the registers that are used to access the port in hardware. A runtime library is used for sending and receiving data over the port. The operations supported by the library are Read(adr, x), Read(adr, x, res) and Write(adr, x). The two implementations of Read are used for blocking and non-blocking read, the latter to be able to model non-determinism in streaming applications.

The runtime library also contains the support of other devices such as RS232, LCD, DDR memory etc. Moreover it

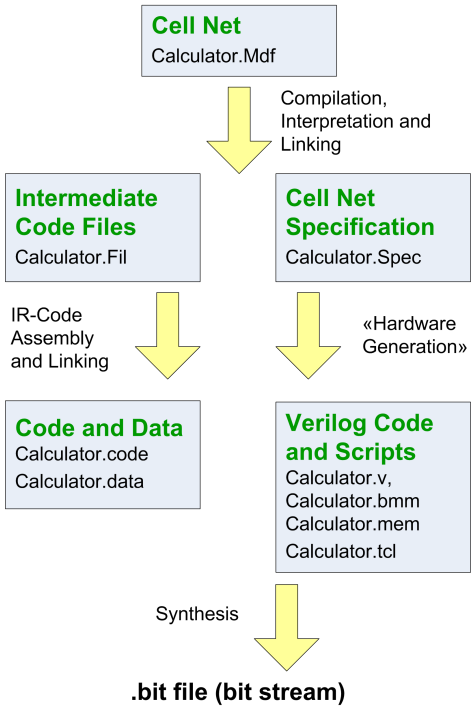

Figure 2. The automated process for mapping an Active Cells program onto an FPGA.

is used to support certain programming language constructs (such as string comparison) and to emulate instructions that are not offered by the hardware component currently in use (such as floating point operations if an FPU is absent).

C. Network Flattening

During compilation, in particular after the network topology is fixed, the network is flattened by the compiler such that no cell contains cell nets any more. This means that in the end all cell nets consist of cells only. In addition, no delegate (virtual) ports are left after flattening. As scopes are lost during this process, some renaming takes place. Flattening all networks is very useful for the hardware generation phase and for an application of the simulator.

D. Simulation

As described above, the compiler generates a hardware specification of the hardware as a textual description of the graph together with the code and data files necessary for execution on the cores in the network. The specification file containing the description of the hardware can also be fed to a cycle-accurate simulator ([18]) to be able to test implementations without actually having to synthesize and download to FPGA hardware. The simulator takes over the role of the synthesizing tool, i.e. it creates instances of the cores, the FIFOs and channels and device controllers and wires them accordingly.

As long as the cycle-accuracy is not of importance, engines that are not yet supported by the simulator can also be formulated in software and be instantiated as processor cores in the simulator. By this, the simulator framework does not have to be adapted for each and every new advent of a new special hardware component and can serve as testing environment very well.

V. Proofs of Concept

To prove the applicability of the Active Cells programming model and its programming environment, several systems have been developed on a single FPGA chip. The granularity of the computation elements ranges from general purpose processor to vector processor to dedicated signal processing engines. In this section, three case studies are presented to illustrate the performance and energy results of three different computer architectures. All of these systems are implemented on a Xilinx ML505 board with a Virtext-5LX50T FPGA chip.

(1) A real-time electrocardiographic (ECG) signal analysis systems. This ECG system performs the analysis of the electrical activity of the heart, including detection of the waves, analysis of their morphology, heart rate variability (HRV) analysis, detection and classification of disease. The analyzed signal represents a standard set of 8 physical channels recorded by a conventional mornitoring device with a sampling rate of 500 Hz [5].

(2) A motion detection system. In this system, a dedicated TRM processor loads input images from compact flash (CF) card and stores them to DDR2 memory. Then the data from DDR2 is loaded and processed in parallel by vector processors. The processing results are streamed back to the DDR2. Using the same I/O process, the resulted data from DDR2 are stored to CF for checking correctness on a PC [6].

(3) A real-time edge detection system. In this system, a VGA input controller samples input RGB video stream at a pixel speed of 65MHz. Each RGB pixel is converted into a gray scale pixel by a one-dimensional filter that implements RGB-to-Y matrix computation. The gray-scale pixels are filter in parallel via a Sobel X Gradient Filter and a Sobel Y Gradient filter to compute the horizontal and vertical gradients in the source video stream. The absolute values of the horizontal and vertical gradients are summed up to approximate the magnitude of the gradients. A multiplier and a comparator following the magnitude function are used to generate a 1-bit mask to indicate if the pixel is on the edge of an Object. This mask will be attached to the source video stream to display the results on a LCD display via a DVI controller in real-time.

Figures 3 and 4 give the corresponding computer architectures for two of the systems. The systems have very different hardware architectures, in particular the computation granularity among them is very different. The ECG system has homogeneous architecture and uses a slow but more flexible general purpose TRM as the computation unit. The Motion Detection system has a heterogeneous architecture that involves vector processors and general purpose processors. The Edge Detection system uses engines implemented in pure hardware as computation units, and only uses a TRM to control the I2C bus to set up the registers in the VGA and DVI encoders. Table I reflects the results in terms of data bandwidth, size and power consumption. It is clear that multiple hardware engines architecture provide better performance and energy efficiency. However, this computer architecture is severely application dependent. Depending on the time-to-market and programmability condition, the system designers can quickly model and measure the system and decide which system architecture meets the performance and energy requirements better.

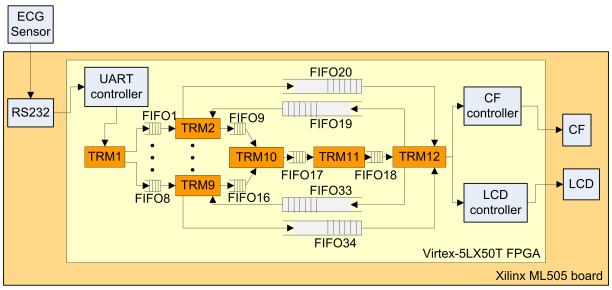

Figure 3. Computer architecture of the ECG system

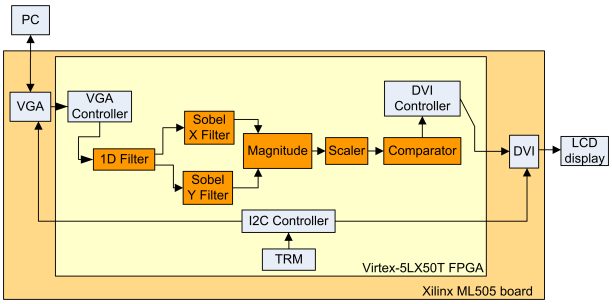

Figure 4. Computer architecture of the Edge Detection system

Table I
Performance, size and power consumption results of above three systems

System	Data bandwidth (bits/second)	Size	Power consumption (Watt)
ECG	64K	48% LUTs, 86% BRAMs, 25% DSPs	0.978
Motion Detection	186.624M	56% LUTs, 93% BRAMs, 70% DSPs	4.21
Edge Detection	1.536G	6% LUTs, 13% BRAMs, 47% DSPs	0.766

VI. Conclusion

It is challenging to model various systems with different architectures and performance or energy requirements in a unified way. And it is even more challenging to automatically *produce* such a targeted running system based on an abstract and unified programming model.

In order to provide a modeling environment that has such capabilities, a full understanding of the class of targeted applications and of hardware-, language- and compiler-design is necessary. Our computing model is based on such an understanding. The results of the case studies and ongoing working experience (and even its use in commercial product development) have proven that the Active Cells computing model is very well applicable to real-world problems with a short time to market.

Our approach is not about the sole translation of programs to circuits but rather provides a way to combine a high level synthesis with an acceptable overhead with respect to usage of resources and energy consumption on an FPGA, similar to the introduction of high-level programming languages replacing pure assembler code some decades ago. To the best of our knowledge such an approach does not exist yet.

An extension of the Active Cells computing model to address the automatic system construction for multi-chip distributed multi-core systems is planned.

Acknowledgment

A substantial part of the work has been funded by Microsoft in the project 'Supercomputer in the Pocket' in the Microsoft Innovation Cluster for Embedded Software. The authors would like to thank Florian Negele and Paul Reed for fruitful discussions and valuable suggestions. Furthermore we would like to express our gratitude to Alexey Morozov and Patrick Hunziker from the University Hospital in Basel for their contribution to the case studies.

References

[1] N. Wirth, "The Tiny Register Machine (TRM)," ETH Zürich, Computer Systems Institute, Tech. Rep. 643, 10 2009.

[2] L. Liu, "A 12-core processor implementation on FPGA," ETH Zürich, Computer Systems Institute, Tech. Rep. 646, 10 2009.

[3] L. Liu, "A bus-based on-chip message passing network," ETH Zürich, Computer Systems Institute, Tech. Rep. 645, 10 2009.

[4] N. Wirth, "A Token-Ring for the TRM," ETH Zürich, Computer Systems Institute, Tech. Rep. 647, 10 2009.

[5] L. Liu and O. Morozov, "A process-oriented streaming system design paradigm for FPGAs," in *Proceedings of the 2010 International Conference on Reconfigurable Computing and FPGAs*, ser. RECONFIG '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 370–375. [Online]. Available: http://dx.doi.org/10.1109/ReConFig.2010.39

[6] L. Liu, O. Morozov, Y. Han, J. Gutknecht, and P. Hunziker, "Automatic soc design flow on many-core processors: a software hardware co-design approach for fpgas," in *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, ser. FPGA '11. New York, NY, USA: ACM, 2011, pp. 37–40. [Online]. Available: http://doi.acm.org/10.1145/1950413.1950424

[7] J. B. Dennis, "First version of a data flow procedure language," in *Programming Symposium, Proceedings Colloque sur la Programmation*. London, UK: Springer-Verlag, 1974, pp. 362–376. [Online]. Available: http://dl.acm.org/citation.cfm?id=647323.721501

[8] W. M. Johnston, J. R. P. Hanna, and R. J. Millar, "Advances in dataflow programming languages," *ACM Comput. Surv.*, vol. 36, pp. 1–34, March 2004. [Online]. Available: http://doi.acm.org/10.1145/1013208.1013209

[9] G. Kahn, "The semantics of a simple language for parallel programming," in *Information processing*, J. L. Rosenfeld, Ed. Stockholm, Sweden: North Holland, Amsterdam, Aug 1974, pp. 471–475.

[10] C. A. R. Hoare, "Communicating sequential processes," *Commun. ACM*, vol. 21, pp. 666–677, August 1978. [Online]. Available: http://doi.acm.org/10.1145/359576.359585

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

[12] C. Hewitt, P. Bishop, and R. Steiger, "A universal modular actor formalism for artificial intelligence," in *Proceedings of the 3rd international joint conference on Artificial intelligence*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1973, pp. 235–245. [Online]. Available: http://dl.acm.org/citation.cfm?id=1624775.1624804

[13] L. Bläser, "A component language for pointer-free concurrent programming and its application to simulation," Ph.D. dissertation, ETH Zürich, 2007.

[14] P. J. Muller, "The active object system design and multiprocessor implementation," Ph.D. dissertation, ETH Zürich, 2002.

[15] P. R. C. Reali, "Using oberon's active objects for language interoperability and compilation," Ph.D. dissertation, ETH Zürich, 2003.

[16] F. Friedrich and J. Gutknecht, "Array-structured object types for mathematical programming." in *JMLC*, ser. Lecture Notes in Computer Science, D. E. Lightfoot and C. A. Szyperski, Eds., vol. 4228. Springer, 2006, pp. 195–210.

[17] F. Friedrich, J. Gutknecht, O. Morozov, and P. Hunziker, "A mathematical programming language extension for multilinear algebra." in *Proc. Kolloqium über Programmiersprachen und Grundlagen der Programmierung, Timmendorfer Strand*, 2007.

[18] R. Stoll, "Development of a multicore simulator framework," Master Thesis, ETH Zürich, 2010.