Erlang inspired Hardware

Paulo Ferreira
CIETI–Laboris/Instituto Superior de Engenharia do Porto
Email: paf@dei.isep.ipp.pt

João Canas Ferreira
DEEC/Faculdade de Engenharia Universidade do Porto
Email: jcf@fe.up.pt

José Carlos Alves
DEEC/Faculdade de Engenharia Universidade do Porto
Email: jca@fe.up.pt

***Abstract*—The Erlang programming language is a concurrency oriented functional language, based on the notion of independent processes and uses message passing for communication between processes. It is specially adapted to the realization of highly reliable distributed systems. In this paper it is analyzed the use of the Erlang's computational paradigm for the design and implementation of application specific heterogeneous computational systems. The main objective is to use for the low level implementation the same computational model used in high level view of the system. This will allow an easier and faster design space exploration and optimization.**

I. Introduction

The functional language Erlang [1] is based on the notion of concurrent independent *processes*, communicating by message passing, thus not sharing any common data. The *processes* are explicitly specified by the programmer, capturing the natural parallelism of an application into different processing tasks that interact by passing messages (see figure 1). An Erlang inspired computing architecture is interesting, because it can be used to provide system builders with an heterogeneous computing model, suitable for modular Field Programmable Gate Array(FPGA) implementations.

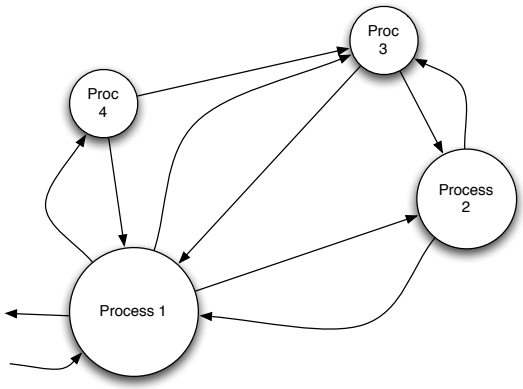

Fig. 1. Erlang system with 4 processes and message passing

The Erlang programming model can bridge the differences between the low level computational model implementation and the high level model available in the programming language. The *processes* can be allocated to heterogeneous hardware structures with different speed-area tradeoffs, according to the specific requirements of the application. Present FPGA devices offer a rich set of resources that can be explored to map the set of processes of an application, either statically or dynamically.

This PhD work proposes a design framework targeting complex FPGA devices, to automate the design space exploration for applications specified in an Erlang language subset. This is accomplished by mapping into heterogeneous hardware structures (eg. FSMDs–Finite State Machines with Datapath, ASIPs–Application Specific Instruction Set Processors, or even embedded processors) the set of computing processes and communication links that implement the desired application.

II. Related Work

There are many works considering the use of functional languages for parallel programming [3], analyzing and proposing the use of special hardware for the execution of functional languages [4], and functional languages have also been used for hardware description has been discussed [5]. The cited works (and many others) are very important, but they seldom associate the three subjects — functional languages, concurrency and hardware models — at the same time. Besides, in previous works concurrency in functional languages is only supported by task switching in conventional processors.

A previous work proposed using a single custom processor for running all the processes of an Erlang system [2], but making hardware to run Erlang could be done in a different way, mapping each Erlang *process* to an autonomous hardware computational structure, communicating with other *processes*, by receiving and sending messages.

As the focus of this work is on the concurrent aspects of the Erlang language, the computation models of Occam/CSP and derived languages and their application to hardware [6] are also important. When compared with CSP/Occam [11] the parallelism model of Erlang is simpler. There is only concurrency among processes, not inside each process. So the degree of parallelism of the system, is a direct function of the number of processes of the system, and is easily controllable. Inside each Erlang process, the execution is purely sequential, so more "normal" to program. As Erlang is a functional language, the data dependencies are clearer, allowing a straightforward hardware description and optimization.

In Bluespec [12] there are also "guards", with an use and meaning very similar to Erlang guards, but scheduled in a very different way. Bluespec is based on guarded atomic actions (or rules), composed by a guard (a predicate) and a body that updates the system state in an atomic action. The scheduling of

all the rules is made by the very sophisticated synthesis tool, with possible hints from the hardware designer. In contrast, the Erlang functions have a simple sequential ordering, with each guard being examined in a sequential order.

While not allowing the sophisticated optimizations of Bluespec, the sequential ordering of the Erlang execution inside each process, is more understandable for a "software oriented programmer". As each process is *isolated* from the others, the optimizations can be process-specific, simplifying the usual iterations in the hardware design workflow.

III. Hardware support for Erlang

The messages are the "connection" between the different hardware blocks or *processes*. Thus, a process may have different implementations, as long as the messages and the associated protocol are not changed, in order to satisfy the specific computational needs of the global system. From a general purpose processor to a dedicated circuit, the computing power and logic complexity of each *process* can be finely tuned.

The design space exploration is simplified because the *processes* are independent and common to the software (high level) and hardware (low level) views of the full system, providing a natural multi-layer *partitioning* strategy to the optimization of the system. The communication link architecture can also be changed, globally or only for some parts of the system, providing the required communication paths among the *processes* are maintained.

This can provide support for transparent software to hardware, or hardware to software migration. A software *process* can use an hardware helper for performance reasons, and an hardware *process* could use a software helper (in an available processor) for non time constrained tasks, if support for hardware-software messaging is made available.

A. Sub-areas

1) Compilation: The ideal tool would convert Erlang to optimized hardware, but the first step is the elaboration a model for the conversion of an Erlang-like language to hardware, using some restrictions. The use of the Core Erlang intermediate representation is a good strategy because Core Erlang is simpler and more regular than Erlang, but is still a functional language, simplifying certain program analysis tasks [7].

Due to the highly dynamic nature of Erlang, some language features like dynamic process creation will be difficult to implement, but even excluding those features, the expressiveness of the language is very good. The dynamic creation of processes is an interesting research topic for applying the dynamic reconfiguration features of modern FPGAs.

2) Communication : In an Erlang application, any process may communicate with any other process. The communication in the system can be done using Networks-on-Chip, shared buses or point to point connections. NoCs are thought almost always with ASICs on mind, where point to point connections are expensive, and even when used, they can be optimized by point to point shortcuts [8].

In FPGA implementations the routing costs are not as high as in ASICs, because there are already in place connections to all the FPGA blocks. But, restricting the destinations of the messages from a process, one can elaborate the system with a series of point to point buses, using the appropriate number of bits, to balance the performance of the communication and the routing resource usage.

3) State and recursion issues: In Erlang, a process is implemented with a recursive function, one of the pillars of functional programming. However, to implement them in hardware, they must be converted first into an iterative version [9]. This transformation is simplified if recursive functions are made to be tail-recursive, and then the function can be implemented by an FSMD using the arguments of the function as the state of the FSMD. This way, the issues around FSMD implementation of recursive functions [10] are eliminated.

4) Computational Structures: An Erlang function is written as set of clauses, each clause may have a "trigger" condition or *guard*, and will be executed if the trigger condition is satisfied. In order to create an hardware model suitable for the "execution" of an Erlang function, first all the conditions are made explicit. Then each clause depends of just one condition, and following the Erlang execution model, the first condition to be true triggers the execution of the associated code. The `gcd` function, a function that finds the greatest common divisor, serves as small example:

```
gcd( A, B ) when A < B  -> gcd( B, A );
gcd( A, B ) when B == 0 -> A;
gcd( A, B ) when 1      -> gcd( A-B, B );
```

After this, the function execution can be mapped into a state machine with two state variables S0 and S1, representing the two arguments of the Erlang function. So a template model to execute the *gcd* Erlang function can be done like in figure 2, where the conditions are examined one at a time (from top to bottom). After the execution of one action, the testing of the conditions restarts from the first one. The symbol *end* means the computation has ended, and the result is available on *out*.

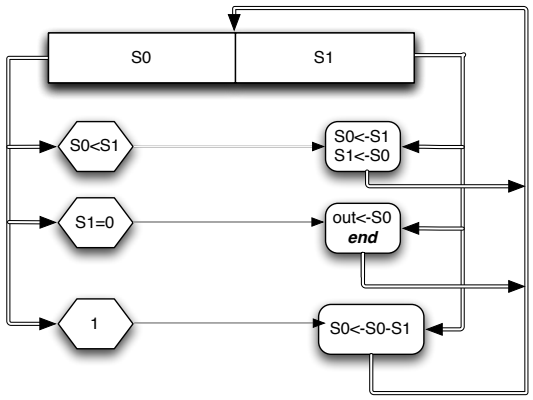

Fig. 2. GCD function execution template

This can be generalized, with conditions being evaluated in parallel if a priority selector is used to enforce the proper sequence, resulting in the diagram shown in figure 3.

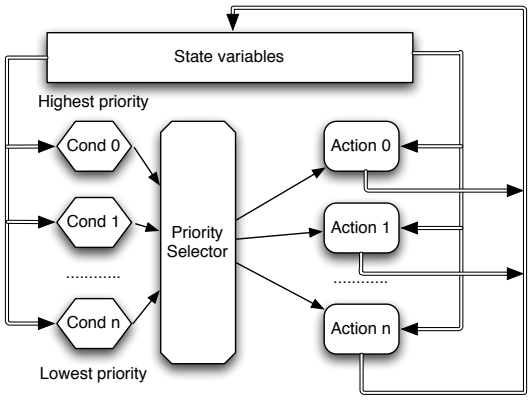

Fig. 3. Generic optimized function execution template

Inside each process, there are several alternatives for the implementation of each action, from a combinational circuit, to a custom microprocessor. Each action can also be implemented by another similar FSMD structure (a call to other function).

The sequential ordering of the execution inside each process implies that the actions are mutually exclusive. When one action is active, all the others of the same process are inactive. This allows hardware sharing among different actions inside one process. As there is no need to preserve any kind of state between two invocations of the same action, this opens the possibility of hardware sharing between actions from different processes.

IV. Current status

A series of simple prototypes have been implemented and simulated, using a custom preprocessor to generate a Verilog module from the FSMD definitions. A simple bus capable of interconnecting the modules was defined, with an associated protocol. The prototypes were used to refine a fully parameterized FSMD model of an Erlang-like process, integrated with a command line, script oriented workflow.

The debugging of the prototypes has been easy, due to the small *distance* between the Erlang source code and the Verilog state machines. The implementation of an automatic translator is being done, but is dependent from a study of memory architecture implementations, in order to give the system elaborator a full set of memory alternatives, adapted to the FSMD model and to the FPGA's internal architecture.

These concepts and tools are being applied and tested in the design of an hardware equivalent of a subset the Erlang *crypto* library. This case study was selected because *crypto* is a very used Erlang library that currently uses the OpenSSL C library, involves large integer computations and is very useful in embedded network systems. The implementation is being done making Erlang functions, testing the Erlang functions, transforming the Erlang functions to have a simple Verilog implementation, and then translating the Erlang functions into Verilog implementations, with the aid of an automatic script that generates the necessary Verilog code from the original conditions and actions.

The future work will include support for:

- Choosing the type of implementation for each process and action
- Parameterizing the connections (type, width) between the processes
- Fully automatic translation from a subset of Erlang into Verilog

This will allow the creation of a set of tools for design space exploration, useful for validating all the available alternatives and the easy creation of application specific heterogeneous computational systems. With a solid set of tools and the knowledge of the compromises available from design space exploration, the investigation of dynamic Erlang hardware process creation will be possible.

References

[1] J. Armstrong, *Programming Erlang*. Raleigh, NC: Pragmatic Bookshelf, 2007.

[2] R. Tjärnström and P. Lundell, "Ecomp - an erlang processor," in *Sixth International Erlang/OTP User Conference*, Älvsjö, Sweden, 2000. [Online]. Available: http://ftp.sunet.se/pub/lang/erlang/euc/00/processor.ppt

[3] D. D. Chamberlin, "Parallel implementation of a single-assignment language," Ph.D. dissertation, Electrical Engineering Department Stanford University, Stanford, CA, USA, 1971.

[4] W. Stoye, "The implementation of functional languages using custom hardware," Ph.D. dissertation, Computer Laboratory, University of Cambridge, Cambridge, December 1985.

[5] A. Mycroft and R. Sharp, "Hardware/software co-design using functional languages," in *Proceedings of TACAS 2001 – Tools and Algorithms for Construction and Analysis of Systems*, 2001, pp. 236–251. [Online]. Available: http://rich.recoil.org/publications/tacas01.pdf

[6] I. Page and W. Luk, "Compiling occam into field-programmable gate arrays," in *Oxford Workshop on Field Programmable Logic and Applications*, W. Moore and W. Luk, Eds. Abingdon: Abingdon EE & CS Books, 1991, pp. 271–283. [Online]. Available: http://www.doc.ic.ac.uk/~wl/papers/fpl91a.pdf

[7] R. Carlsson, B. Gustavsson, E. Johansson, T. Lindgren, S.-O. Nyström, M. Pettersson, and R. Virding, "Core Erlang 1.0.3 language specification," HiPE Research Group – Uppsala University, Tech. Rep., 2004. [Online]. Available: http://www.it.uu.se/research/group/hipe/cerl/doc/core_erlang-1.0.3.pdf

[8] U. Ogras, R. Marculescu, H. G. Lee, and N. Chang, "Communication architecture optimization: making the shortest path shorter in regular networks-on-chip," in *Design, Automation and Test in Europe, 2006. DATE '06. Proceedings*, vol. 1, March 2006, pp. 6 pp.–.

[9] J. Arsac and Y. Kodratoff, "Some techniques for recursion removal from recursive functions," *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 2, pp. 295–322, 1982.

[10] V. Sklyarov and I. Skliarova, "FPGA-based implementation and comparison of recursive and iterative algorithms," in *Proceedings of the 15^{th} International Conference on Field-Programmable Logic and Applications - FPL'2005*, T. Rissa, S. J. E. Wilton, and P. H. W. Leong, Eds. Tampere, Finland: IEEE Press, 8 2005, pp. 235–240.

[11] C. A. R. Hoare, *Communicating Sequential Processes*. Prentice Hall, 1985. [Online]. Available: http://www.usingcsp.com/cspbook.pdf

[12] D. L. Rosenband and Arvind, "Hardware synthesis from guarded atomic actions with performance specifications," in *IEEE/ACM International Conference on Computer-Aided Design, 2005. ICCAD-2005.*, San Jose, 2005, pp. 784–791.