Tail recursion in Hardware

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

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

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

***Abstract*—High performance embedded computing systems can benefit from parallel architectures built around heterogeneous computational entities. The computing paradigm based on independent process communicating by message passing is an interesting model for custom hardware implementation because each process may be mapped into independent and dedicated processing blocks, thus easing the design space exploration task.**

This computing model is the basis for some functional programming languages relying heavily on recursion as a powerful, yet simple, tool to specify algorithms. The implementation of recursion in custom hardware has been problematic due mainly to the unpredictable requirements for the stack-like memory structures.

In this paper we propose the transformation of recursive specifications into *tail recursive*, easing the hardware implementation and avoiding the need for complex stack memory structures. This opens the way to the utilization of functional programming languages (eg. Erlang) as a unified approach for the specification of both functional software and custom computing hardware blocks.

I. Introduction

Performance reasons are the main justification for exploiting parallelism in custom designed computing systems. In this scenario, resource utilization is the main motivation behind heterogeneous systems. With FPGA based systems, a problem-specific or instance specific approach can be used instead of a generic computing architecture, because the FPGAs can be reconfigured with the exact computing resources needed. In a problem specific architecture, it is convenient that the system designer will specify the different tasks, and the degree of parallelism of the system. As a result of specifying the different tasks, these can be efficiently mapped into an heterogeneous system, customized to the specific problem and to the FPGA's available resources.

The underlying architecture can be hidden or can be exposed by the chosen programming model. If the architecture is exposed by the programming model, the software hardware interface may be simplified. So instead of making hardware to fit a programming language, a better solution is to use a language that fits the underlying architectural model. So, an ideal language for describing an embedded system would be a concurrency oriented language with explicit parallelism, with a programming model similar (or suitable) to a low level FPGA implementation.

In Asanović et al [1] the general issues about parallel computing are presented in an organized way, as being:

- Applications
  1. What are the applications?
  2. What are common kernels of the applications?
- Hardware
  3. What are the hardware building blocks?
  4. How to connect them?
- Programming Models
  5. How to describe applications and kernels?
  6. How to program the hardware?
- Evaluation:
  7. How to measure success?

On embedded high performance computing, the middle items (hardware and programming models) have a greater importance than in general parallel computing. But the issues surrounding them could be simplified if the hardware architecture has a strong resemblance with the programming models, justifying the approach proposed in this paper.

II. Erlang

Erlang is a functional concurrency oriented language [2], useful for programming distributed applications. In Erlang, a system is built with concurrent independent processes (an independent computational abstraction), communicating only by message passing (see figure 1). While running an Erlang software application, the processor of the host computer is time shared by all the processes. One of the characteristics of Erlang run-time system is the fast switching between processes, and the support of a great number of processes, when compared with other concurrent languages.

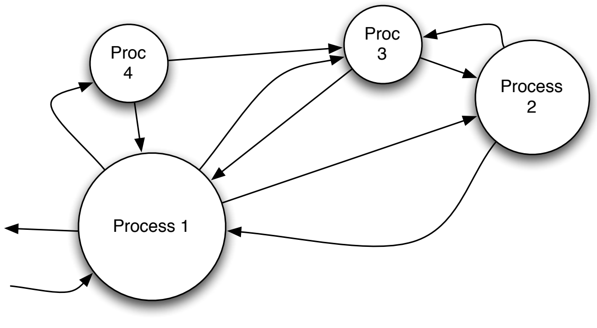

Fig. 1. Erlang system with 4 processes and message flow

Erlang has an interesting computational model for implementation in custom hardware. Each "process" may correspond to an hardware "module", translating from a "CPU time-sharing" model to a fully parallel and heterogeneous execution model. This will bring as consequence, the use of the similar paradigms for the high-level architectural description and for the low-level implementation, without a significant semantic gap.

The message passing paradigm is being adopted in many other parallel programming frameworks, even when the lower level support mechanism is shared memory [3], in order to avoid data races. In an FPGA based system, it has the advantage of avoiding global data sharing and the need for global buses.

III. Erlang to FPGA mapping

A. Heterogenous Parallelism

In FPGA implementations, heterogeneous hardware parallel system are preferred, in order to optimize resource usage. This can be achieved with the mapping of each Erlang process to a specific hardware block, highly customized to the needs of that process. This approach, to be useful, must be supported from the high level system description down to the hardware level.

Not sharing resources among different processes may seem a waste of resources, but the number of processes and their complexity is specified by the programmer. Also on each hardware block, the specific implementation of each process can be tuned from a resource saving to an performance oriented approach. The heterogeneous architecture opens the possibility of varying each process implementation, in order to perform an exhaustive design space exploration.

The absence of automatic parallelism extraction can be advantageous in some cases, because it means the system's designer is in full control, to give the system the right amount of parallelism where it is needed.

B. Message passing

The message passing paradigm is one of the great advantages of an Erlang system. Avoiding data-sharing issues, and providing process isolation, the messages are at the same time, the "glue" and the isolation among the different processes or hardware blocks. As long as the same messages are delivered, the communication channels and protocols can be changed, and implementation of other processes can also change. The message passing infrastructure can be implemented in hardware using Networks-on-Chip [4], point to point connections, buses or any combination of these alternatives.

C. Process architecture

The architectural implementation of each process must consider the various possible constraints, from computational performance to FPGA resource usage. For more complex and non time-constrained tasks, a microprocessor could be used, but for simple or time-constrained tasks, an FSMD (Finite State Machine with Datapath) implementation may be desirable.

In the last case, it is needed to map an Erlang process into an FSMD model. There are two apparent conflicts to solve here. Erlang is a functional (single assignment), no state language and FSMDs are State Machines. An Erlang process should be written in a recursive form, while an FSMD has an iterative model of computation.

Other studies about custom hardware for running functional languages are not focused these problems, because they try to find a generic architecture to run all the code [5], not a specific architectural implementation for each process. An Erlang processor was built by Tjärnström and Lundell [6], but on that processor the concurrency was achieved by task switching.

The studies of functional languages for hardware description have been prolific [7], but they are focused on synthesizing correct hardware and verifying hardware, using very high level hardware descriptions [8].

IV. FSMD model

A. Recursion in Hardware

Usually recursion in programming languages is based on RAM based stacks, with a hardware implemented stack pointer, in the majority of modern processors. In some embedded processors, the stack is internal and limited in size, bringing limitations to subroutine use and recursion use.

In the common hardware description languages there is no support for recursive definitions of computational structures [9], and the support of recursive functions is impossible. This is true even in languages made for people with software expertise [10] like Handel-C.

Recently there is a surge of interest on the use of recursive algorithms in hardware. Sklyarov has deeply studied the subject [11], and has several proposals comparing recursive and iterative implementations [12], adapting hierachical FSMs and doing parallel implementations of hierarchical FSMs. He uses an auxiliary stack for preserving the state of the state machine, another stack for the "modules", and another stack for data, like in figure 2. The "modules" are "subsets" of all the required states and are used for simplifying the overall elaboration of of the circuit.

In the majority of the cases the recursion is done with a stack for the current state of the state machine (and module), and another stack to keep the variables [13]. The only case where a stack is not needed uses dynamic reconfiguration [10], although the reconfiguration times may compromise the performance in some architectures.

The use of a stack of variable size is common and easy in software, as modern general purpose processors support indirect addressing, and the use of a system stack. In hardware, the maximum size of the stack must be known at synthesis time, and specified in the hardware description language used to describe the circuit. So, a small stack can be a limitation to the size of the problems solved, and a large stack is a waste of internal RAM in a reconfigurable circuit.

The internal architecture of FPGAs is sometimes used to implement FIFOs and other kinds of shift registers, but for

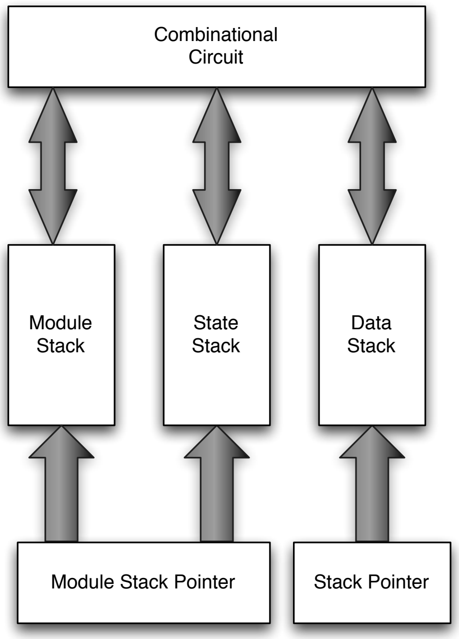

Fig. 2. Recursive FSMD implemented with stacks

implementing a stack the use of internal RAM and a indexing register is the only possibility [14]. This means the size of the internal RAM blocks must be carefully chosen (if possible!), and the access to the implemented stacks involves always a previous access to the indexing register used.

B. Recursion in Erlang

An interesting variant of recursion called tail-recursion is used in many functional and logic languages [15]. The advantage of tail-recursive functions is the ability of running a recursive function in a constant space, if the compiler (or interpreter) has support for it. One can see how tail recursion works with a simple example in the Erlang language [2]:

```
factorial(1)->1;
factorial(N)->N*factorial(N-1).
```

Listing 1: Factorial in the Erlang language

Rearranging the code of Listing 1, but keeping the recursion it is possible to obtain a *tail-recursive* version, using *accumulators*.

```
factorial(N)->factorial2(N,1).
factorial2(1,F)->F;
factorial2(N,F)->factorial2(N-1,F*N).
```

Listing 2: *Tail-recursive* factorial

On listing 2 there are two functions, `factorial` and `factorial2`. The last one uses an additional argument called *accumulator*, where is stored the value that is being "*built*". One can also see that `factorial2` is called recursively, but without any pending calculations, on the last line of listing 2. As there are no pending calculations, the language can reuse the same memory space for the next call, without allocating additional space for each recursive call.

This modification is very useful for making a state machine version of the algorithm, that can be easily in implemented in hardware. The arguments of the function are used as the state variables. The relation between the original function arguments and the arguments when the function is called recursively, gives the expression relating $State_n$ with $State_{n+1}$ of the state machine.

An example of this transformation can be seen in the flowchart given in figure 3, with Ai corresponding to the ith argument of the function.

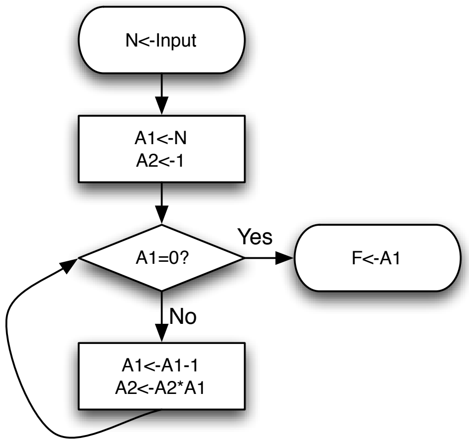

Fig. 3. Factorial flowchart

The advantages of tail recursion are obvious, because it has all the advantages of an iterative algorithm (simplicity and constant memory space) and all of the advantages of recursion (simplicity and easier formal analysis).

The proof that all recursive algorithms can be transformed into a tail recursive formulation can be found on Ward [16], and further developments on recursion removal and introduction are exposed in [17]. So, a recursive algorithm can be transformed in an iterative (or tail-recursive) algorithm, and an iterative algorithm can be also (in most cases) transformed into a recursive algorithm.

C. Tail Recursive Hardware

Transforming recursive functions into tail-recursive functions, to map into hardware, eliminates the need of a stack, stack sizing or dynamic reconfiguration, because the algorithm can be implemented in an iterative form, in constant memory space. On a normal FSMD there is a combinational circuit and the state variables like in figure 4, but with a recursive FSMD the state registers (at least) need to be replicated on a stack of a certain depth like in figure 5.

To exemplify the proposal, it is used the algorithm for calculating the N-th element of the Fibonacci series. Due to it's tree-like growth, Fibonacci's algorithm is a classical example of complexity in recursive functions, being used also

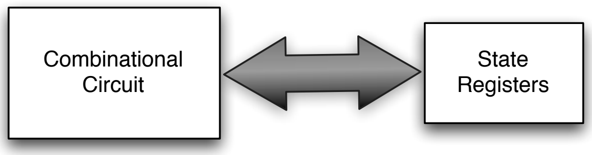

Fig. 4. Normal FSMD

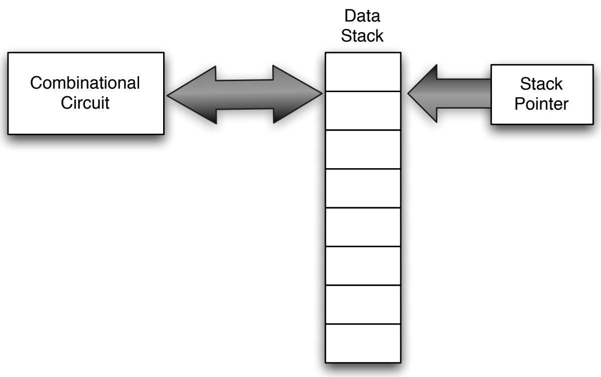

Fig. 5. FSMD with a stack

in Ferizis and El Gindy [10], to study the implementation of a recursive function in hardware.

Listing 3 shows a classical Fibonacci recursive function (coded in Erlang). It involves a double recursion (two recursive calls for each number), but can be transformed into tail-recursive with the use of two accumulators (auxiliary arguments).

The idea behind the transformation is to calculate the Fibonacci numbers, in reverse order starting with the Fibonacci of zero. The first argument of the `fib_iter` function works as a down counter, reaching zero after N calls to the function. On each call to the function, it is calculated the next Fibonacci number, adding the current result to it, and replacing the result with the previously calculated value.

The corresponding flowchart can be seen on figure 6, again with Ai corresponding to the ith argument of the function.

```
fib(0) -> 0;
fib(1) -> 1;
fib(N) -> fib(N-1) + fib(N-2).
```

Listing 3:Recursive Fibonacci algorithm

```
fib(N) -> fib_iter(N, 0, 1).

fib_iter(0, Result, _Next) ->
  Result;
fib_iter(Iter, Result, Next) ->
  fib_iter(Iter-1, Next, Result+Next).
```

Listing 4: Tail Recursive Fibonacci algorithm

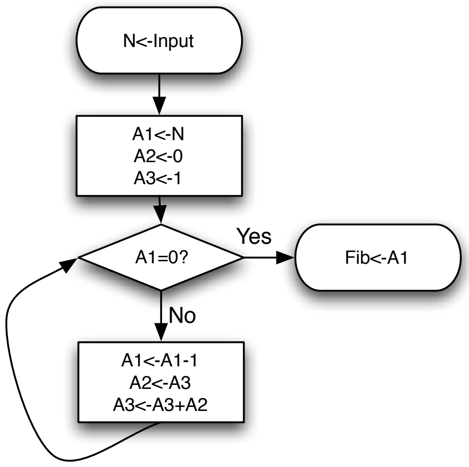

Fig. 6. Fibonacci flowchart

With this simple transformation the need for a stack is eliminated, but it also transformed the *process growth rate* of the Fibonacci function. Ferizis and El Gindy [10] call *process growth rate* to the ratio between the work of a *child*, and the work of it's root, and abandon the Fibonacci function in their implementations, because in the normal recursive form (non tail-recursive) each Fibonacci number calculation, provokes recursively two calculations, imposing a tree-like computational structure.

So, the tail recursion transformation, besides allowing the use of recursion in a constant space, in some cases, also lowers the computational complexity of the algorithms.

Compared to previous approaches, instead of a stack, there is only the need to provide additional space for the *accumulator* variables like in figure 7, saving FPGA resources compared to a stack, and avoiding the dilemmas of stack sizing.

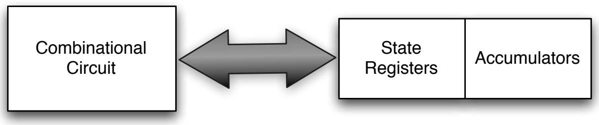

Fig. 7. Tail-recursive FSMD

D. Detailed model

Using the tail recursive FSMD, and hardware model, the implementation of a generic Erlang function can be made. The implementation of an Erlang process is similar, because Erlang processes are implemented with tail recursive functions [2], and the message passing primitives.

An Erlang function is made from different clauses, each with optional *guard conditions* (`when A<B` in Listing 5). The guard condition may be implicit or explicit. For execution of a function, each guard condition is evaluated in order, and when a true condition is found, the corresponding actions are executed. Many times (in a tail recursive function) the action

is just calling the same function with a new set of arguments. In listing 5 there is an example of an GCD (Greatest Common Divisor) function in Erlang.

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

Listing 5: Greatest Common Divisor

In order to generalize the model, a useful change is can be made, turning all the guards into explicit guards like in listing 6.

```
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 ).
```

Listing 6: Greatest Common Divisor with explicit guards

From this representation, the table I of conditions and states can be obtained, starting from an *initial* state and stopping at *end*, considering S_0 as the first argument of the function and the clauses numbered from 0.

TABLE I
INFO FOR THE FSMD ELABORATION

Clause	Condition	S_0*	S_1*	Next state
0	S_0<S_1	S_1	S_0	initial
1	S_1==0	S_0	S_1	end
2	1	S_0-S_1	S_1	initial

Using this information the elaboration of the FSMD for execution of Erlang like functions can be automated. The FSMD has an initial state, and from the initial state, it sequentially checks the condition of each clause, and if the condition is true, it calculates the next values of the state variables (S_0*,S_1*), and goes to the next state. If a condition is not true, the following clause is tested, as seen on figure 8.

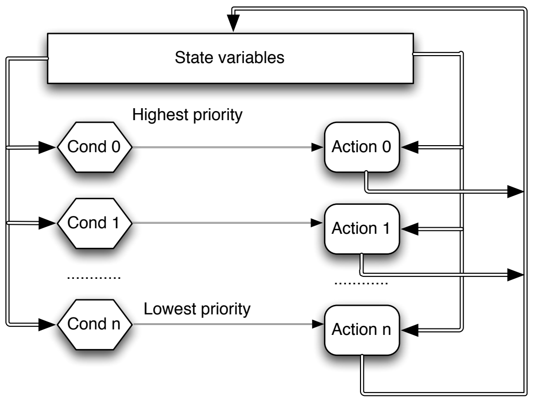

Fig. 8. Generic function execution template

The evaluation of the conditions can generally be made with simple combinational circuits, because in Erlang, only a limited number of built-in functions are allowed in guards by performance reasons [2]. More complex functions can always be used in conditions, if they are evaluated before and their result used as an additional argument.

In this model, the evaluation of the conditions can be optimized. They can be evaluated in parallel, as long as only the correct action is "triggered", using a priority encoder, without indulging in a great FPGA resource usage, as the number of clauses in a typical Erlang function is low. As an example, in four big Erlang systems (OTP, Ejabberd, CouchDB and RabbitMQ) the source files the functions have under 2.2 clauses in average.

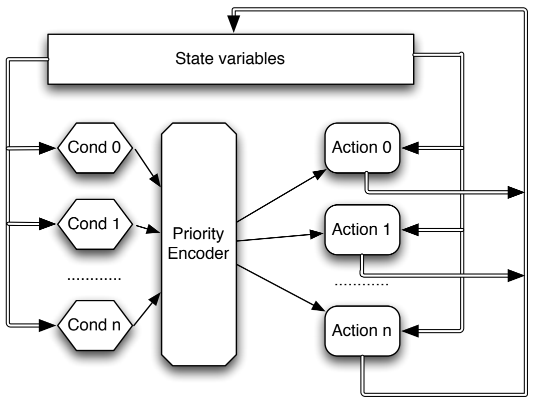

Fig. 9. Generic optimized function execution template

The "actions" have as a result the next values of the state variables, and they have different degrees of complexity. Considering their complexity and also considering the relative execution frequency of each clause, made by profiling the function, an appropriate choice for each "action" circuit can be made. The alternatives go from a purely combinational circuit, to an embedded microprocessor, with an FSMD as middle point between the two.

As the actions are mutually exclusive, the resources used in implementing them can be time shared among the different clauses, merging the circuits implementing the actions, as shown like in figure 10.

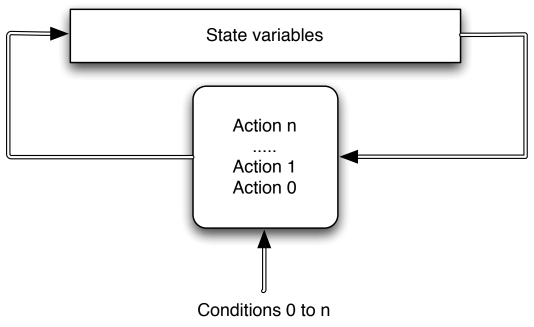

Fig. 10. Circuit sharing the action blocks

The resource sharing optimization is possible even if the implementation of the different actions uses different kinds of circuits (combinational, FSMD or microprocessor), as the priority encoder assures that only one of them is active at the same time.

A version of this template (without resource sharing optimization), was integrated in a custom preprocessor to automatically generate a Verilog module from the fully parametrized FSMD definition. This tool was integrated with a command line, script oriented workflow. When the evaluation of the next value of a state variable is too complex for a combinational circuit, the developed tool allows the instantiation and automatic connection of an additional similar FSMD for that evaluation.

The template was tested with several recursive algorithms (GCD, Fibonacci, binomial coefficients, merge sort, binary search), leading to resource usage and performances of iterative implementations, because tail recursion can be seen as a technique of transforming recursion into iteration of a set of variables (the function's arguments).

TABLE II
RESOURCE USAGE AND CLOCK FREQUENCY FOR SOME TAIL-RECURSIVE ALGORITMS

	GCD	**Fibonnacci**	**Binomial Coef.**
Flip-Flops	71	103	248
4-input LUTs	243	207	455
Slices	122	104	342
Multipliers	0	0	3
Clock Freq. (MHz)	113	153	52

The results presented on table II were obtained using Xilinx ISE 10.1 with the default settings, targeting a Spartan 3E FPGA (xc3s500e-4fg320). The built modules work with 32 bit width numbers. The binomial coefficient module uses an auxiliary division module, with the same tail-recursive structure.

V. CONCLUSION

In hardware implementations, recursion is something to be avoided, because it implied the use of stacks, and stack size dimensioning, or recursion unrolling at synthesis time (when possible). With a tail recursive function, the elaboration of an FSMD is simple, as it was described in this paper. This approach also avoids the need of previous recursion depth determination, as needed in some recursion flattening strategies.

The proposed approach has a great simplicity and does not force a large number of transformations from the algorithmic model to the low level hardware implementation. The conversion of an algorithm to a tail recursive format, is something simple for a programmer familiarized with functional programming, and is done in the initial model elaboration phase. The elaborated hardware has a strong resemblance to the initial model, aiding on the debugging, test and verification phases. The recursive nature of some algorithms should not be seen as something very far away from hardware implementations, but as something that can be easily done in hardware, as it was demonstrated.

The creation of a template for parameterized automatic instantiation of Verilog modules implementing the desired circuits has been done. The parametrization will aid the design space exploration for a specific computing task. The tool is being tested and improved, and will be used in a future workflow for the production of high performance embedded problem specific computing systems, with an heterogenous parallel architecture.

REFERENCES

[1] K. Asanović, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The landscape of parallel computing research: a view from Berkeley," University of California at Berkeley, Berkeley, Tech. Rep. UCB/EECS-2006-183, December 2006. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.pdf

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

[3] S. A. Edwards, N. Vasudevan, and O. Tardieu, "Programming shared memory multiprocessors with deterministic message-passing concurrency: compiling shim to pthreads," in *DATE '08: Proceedings of the conference on Design, automation and test in Europe*. New York, NY, USA: ACM, 2008, pp. 1498–1503.

[4] S. Pasricha and N. Dutt, *On-Chip Communication Architectures*. San Francisco: Morgan Kaufmann, 2008.

[5] S. Vegdahl, "A survey of proposed architectures for the execution of functional languages," *IEEE Transactions on Computers*, vol. C-33, no. 12, pp. 1050–1071, 1984.

[6] 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

[7] M. Sheeran, "Hardware design and functional programming: a perfect match," *Journal of Universal Computer Science*, vol. 11, no. 7, pp. 1135–1158, 2005. [Online]. Available: http://www.jucs.org/jucs_11_7/hardware_design_and_functional

[8] R. Sharp, *Higher-level hardware synthesis*, ser. LNCS 2963. Berlin: Springer, 2004.

[9] P. J. Ashenden, "A comparison of recursive and repetitive models of recursive hardware structures," *Proceedings of VHDL International Users Forum. Spring Conference, 1994.*, pp. 80–89, May 1994.

[10] G. Ferizis and H. El Gindy, "Mapping recursive functions to reconfigurable hardware," in *International Conference on Field Programmable Logic and Applications, 2006. FPL '06.*, Aug. 2006.

[11] I. Skliarova and V. Sklyarov, "Recursion in reconfigurable computing: A survey of implementation approaches," in *Proceedings of the 19th International Conference on Field Programmable Logic and Applications – FPL'2009*, 2009, pp. 224–229.

[12] V. Sklyarov and I. Skliarova, "FPGA-based implementation and comparison of recursive and iterative algorithms," in *Proceedings of the 15th 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.

[13] I. Skliarova and V. Sklyarov, "Recursive versus iterative algorithms for solving combinatorial search problems in hardware," *VLSI Design, 2008. VLSID 2008. 21st International Conference on*, pp. 255–260, Jan. 2008.

[14] P. Alfke, "Creative uses of block RAM," Xilinx White Paper 335, 2008. [Online]. Available: http://www.xilinx.com/support/documentation/white_papers/wp335.pdf

[15] Y. A. Liu and S. D. Stoller, "From recursion to iteration: what are the optimizations?" in *PEPM '00: Proceedings of the 2000 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation*. New York, NY, USA: ACM, 1999, pp. 73–82.

[16] M. Ward, "Proving program refinements and transformations," Ph.D. dissertation, St. Annes College, Oxford, Oxford, UK, 1989. [Online]. Available: http://www.cse.dmu.ac.uk/~mward/martin/thesis/index.html

[17] M. P. Ward and K. H. Bennett, "Recursion Removal/Introduction by formal transformation: An aid to program development and program comprehension," *The Computer Journal*, vol. 42, no. 8, pp. 650–673, Aug. 1999. [Online]. Available: http://comjnl.oxfordjournals.org/cgi/content/abstract/42/8/650