Hardware Design Methodology with the Alpha Language[1]

David Cachera[2], Anne-Claire Guillou, Fabien Quilleré,
Patrice Quinton, Sanjay Rajopadhye, Tanguy Risset
IRISA, Campus de Beaulieu, 35042 Rennes, France
{aguillou, dcachera, quillere, quinton, rajopadh, risset}@irisa.fr

Abstract

We present the Alpha language and its use to design parallel VLSI architectures with the MMAlpha software. The main goal of MMAlpha is to provide a tool-box for compiling parallel architectures on silicon from high level functional specifications. The current version of MMAlpha allows one to obtain in a few hours synthesizable VHDL code for algorithms that can be mapped on linear parallel architectures. The design methodology of MMAlpha is presented, and the main tools available for static analysis of programs, scheduling, verification, code generation, and hardware generation are briefly described.

1 Introduction

Systems on chips (SoC) are heterogeneous architectures including processor cores, Asics, Asips, memories, etc. Designing a SoC requires therefore various kind of tools, each one being targeted to a particular style of design. Ideally, a SoC should be compiled from a high-level specification such as a C program. However, we are far from this goal, as the number of possible design paths is so large that it is currently hopeless to try solving the problem in its full generality.

The work reported here concerns methods for compiling high-level specifications of signal processing algorithms to various target architectures. Based on loop parallelization techniques, it follows the work of Leiserson [1] on retiming and of Lamport [2] on automatic parallelization, and it is supported by *polyhedral model* developed in the 90's [3]. Other projects along the same lines are Pico [4], Compaan [5], and Match [6].

The MMAlpha software that we describe here was developed at Irisa as a transformation tool-box for programs expressed in Alpha [7]. An Alpha program can be interpreted as a set of recurrence equations or, under certain conditions, as the description of a regular VLSI architecture. MMAlpha is a set of Mathematica packages [8] which constitutes a framework for compiling Alpha programs into synthesizable VHDL. Target applications of MMAlpha are in the fields of signal processing [9, 10], image processing [11, 12, 13], and more generally computation intensive algorithms. The aim of this paper is not to present new result, but to present in a concise manner all the researches and implementations that led to the MMAlpha software.

In the remaining of this paper, we summarize the design flow of MMAlpha (section 2), we describe the various tools available in MMAlpha (section 3), and finally we discuss the limitations of this approach (section 4).

[1] This work was partially supported by grant 1802-1 from IFCPAR.

[2] Also with École normale supérieure de Cachan, antenne de Bretagne.

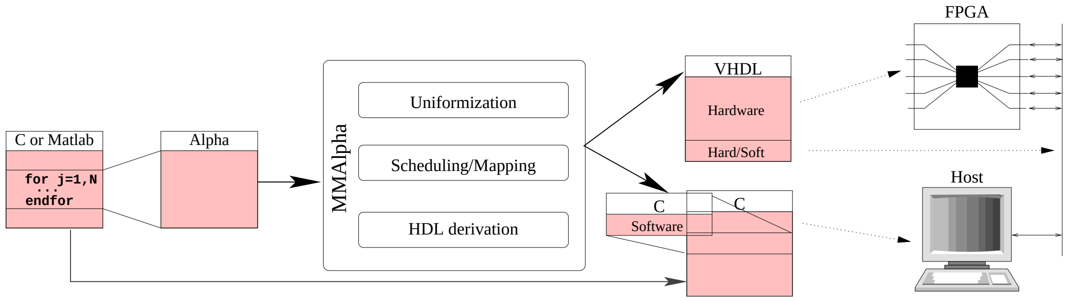

Figure 1: Design flow with MMAlpha. Square shaded boxes represent programs of various languages, and round boxes represent transformations.

2 The design flow of MMAlpha

The design flow with Alpha is illustrated on figure 1. The application is first described in C or in Matlab, and the part which is candidate to be implemented using a parallel architecture is rewritten in Alpha, either manually of using automatic tools. It is then loaded in the MMAlpha software, analyzed, and after the application of rewriting transformations, translated into a combination of VHDL – the hardware part – and of C code – the software interface part. The VHDL can be synthesized either to obtain an Asic or to produce a bit stream for a FPGA architecture.

Based on this, complex computations expressed as several non perfectly nested loops with symbolic bounds values can be parallelized automatically. As shown in [13] for adaptive filtering, one can also build quickly parameterized generators of flexible blocks for parallel hardware implementations.

3 MMAlpha tools

This section highlights some of the main originalities of our methodology. We consider successively the static analysis of algorithms, scheduling, property prover, code generation, and hardware generation.

3.1 Static analysis of algorithms

Alpha is a modular *single assignment* language. This property allows very powerful static analyses of programs to be done. Static analyses of Alpha programs help solving several important problems: verifying some properties – for example, detecting dead code or unassigned values –, estimating some quantities – e.g. the number of calculations performed in a nest of loops as a function of the loop bounds, – or inferring some parameters related to hardware implementations. Recently [14], a powerful technique has been proposed for setting the bit-width of Alpha programs. All these static analyses can be applied in a hierarchical manner to applications written in Alpha.

3.2 Scheduling

Single assignment code makes the detection of potential parallelism easy. The scheduler is actually the central tool of MMAlpha: it allows an Alpha program to be analyzed and scheduled for parallel implementation, either in a bottom-up manner, or after flattening (part of) the hierarchy. The scheduler allows mono-dimensional or multi-dimensional scheduling to be computed, thus providing a large range of solutions to adjust the degree of parallelism to the actual needs of the application. These techniques were illustrated for example on the design of architectures for Kalman filtering [9].

3.3 Proving properties

Most of the transformations implemented in MMAlpha are rewritings that preserve the semantics, thus ensuring an equivalence between the original specification and the final code. Nevertheless, there are many cases when we need to formally prove certain properties that are not ensured by this refinement process. We have developed complementary proof methodologies that use both the MMAlpha environment and the PVS theorem prover [15]. Some properties – for instance, functional equivalence of two Alpha systems defined by means of two distinct algorithms – can be proved by performing MMAlpha transformations. In other cases, we use an interface between MMAlpha and the PVS prover, which enables us to mix MMAlpha transformations steps and PVS induction proofs.

3.4 Code generation and simulation

C code generation from Alpha is needed for simulation purposes, as well as for implementation of some parts of the application on programmable cores. Based on efficient algorithms for scanning polyhedra by loop nests and on memory reuse techniques, Alpha programs can be translated automatically into efficient C programs [16, 17, 18, 19]. This provides a code generator for simulation at any stage of the design flow. The code generator could be very quickly re-targeted in order to generate the interface between the architecture generated from Alpha and its environment [12].

3.5 Hardware generation

The main goal of the MMAlpha environment is to provide a tool-box for fast and safe parallel hardware generation from high level specification. The proposed design flow implements the basics steps of the automatic systolic arrays design methodology, i.e. uniformization of code, scheduling, and hardware derivation. Uniformization allows broadcasts to be replaced by local pipelining of values. Scheduling, mentioned in section 3.2, and hardware mapping (i.e., allocation of computation to processors) are followed by a space-time transformation of code that allows the designer to choose when and where computations take place. Finally, hardware derivation includes automatic generation of control signals, hardware interpretation for all program statements and automatic structuring of the program. The translation produces synthesizable VHDL (compliant with the IEEE–1076.6 standard [20]), after size parameters of the application are fixed. The hardware translator itself is implemented using a syntax directed meta-generator (called Meta-Alpha) which allows fast re-targeting of the generator depending on the type of synthesis tools used.

4 Limitations of MMAlpha

The main limitations of MMAlpha are related to the use of an unconventional language such as Alpha which constitutes a barrier – cultural, more than technical – to the designers. Another limitation is its implementation using Mathematica, a tool with which chip designers are rarely familiar, and which is not often available in their computing environment. This second limitation will be overcome only by rewriting the software using C or C++. As far as the first limitation is concerned, the use of Alpha will have to be hidden to designers, either by using a schematic design capture technique, or more likely, by using C or Matlab as a design entry. Although this possibility is not yet fully implemented, experiments using a MatParser [5] translator have shown that this approach is viable. However, our experience has proved that using directly Alpha as a design language is not technically difficult: indeed, for solving some tasks such as the design of efficient IP generators, it will remain the best approach to build efficient libraries of flexible hardware blocks.

5 Conclusion

We have described the hardware design methodology of MMAlpha based on the Alpha language and we have highlighted some of its mains originalities. This tool is clearly targeted to data intensive parts of processors, and allows generators to be written quickly. Because it strictly sticks to the principle of formally proved program transformations, we believe that the methodology we have presented is a good approach to mastering the complexity of high-level design of SoC. The current version of the MMAlpha software is more a tool-box than a design environment, but recent work has proved its efficiency for generating parallel architectures either for VLSI or FPGA. With the emergence of powerful FPGA co-processors, the need for "FPGA compilers" becomes crucial: MMAlpha can be considered as a prototype of what could be an FPGA compiler using the polyhedral model theory.

References

[1] Charles E. Leiserson and James B. Saxe, "Retiming Synchronous Circuitry," *Algorithmica*, 1991.

[2] L. Lamport, "The Parallel Execution of DO Loops," *Communications of The ACM*, vol. 17, no. 2, pp. 83–93, Feb. 1974.

[3] C. Lengauer, "Loop parallelization in the polytope model," in *CONCUR'93*, E. Best, Ed. 1993, Lecture Notes in Computer Science 715, pp. 398–416, Springer-Verlag.

[4] R. Schreiber et al., "High-Level Synthesis of Non Programmable Hardware Accelerators," in *IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP 2000)*, Boston, July 2000.

[5] B. Kienhuis, E. Rijpkema, and E.F. Deprettere, "Compaan: Deriving process networks from matlab for embedded signal processing architectures," in *8th International Workshop on Hardware/Software Codesign (CODES'2000)*, 2000.

[6] P. Banerjee, N. Shenoy, A. Choudhary, S. Hauck, C. Bachmann, M. Halda, P. Goisha, A. Jones, A. Kanhare, A. Nayak, and S. Periyacheri, "A Matlab Compiler for Distributed, Heterogeneous, Reconfigurable Computing System," in *IEEE Symposium on FPGA as Custom Computing Machines (FCCM 2000)*, B. Hutchings, Ed., Napa Valley, CA, Apr. 2000, pp. 39–48.

[7] C. Mauras, *Alpha : un langage équationnel pour la conception et la programmation d'architectures parallèles synchrones*, Thèse de doctorat, Ifsic, Université de Rennes 1, Dec. 1989.

[8] F. Dupont de Dinechin, P. Quinton, S. Rajopadhye, and T. Risset, "First Steps in Alpha," Tech. Rep. 1244, Irisa, 1999.

[9] A. Mozipo, D. Massicotte, P. Quinton, , and T. Risset, "Automatic Synthesis of a parallel Architecture for Kalman Filtering using MMAlpha," in *International conference on parallel computing in electrical engineering (PARELEC 98)*, Bialystok, Poland, Sept. 1998, pp. 201–206.

[10] A. Mozipo, D. Massicote, P. Quinton, and T. Risset, " A Parallel Architecture for Adaptative Channel Equalization Based On Kalman Filter Using MMAlpha," in *1999 IEEE Canadian Conference on Electrical & Computer Engineering*, Calgary, Canada, May 1999.

[11] E. Mémin and T. Risset, "On the study of VLSI derivation for optical flow estimation," *International Journal of pattern recognition and Artificial Intelligence (IJPRAI)*, vol. 14, no. 4, pp. 441–463, June 2000.

[12] S. Derrien and T. Risset, "Interfacing compiled FPGA programs: the MMAlpha approach," in *PDP-TA2000: Second International Workshop on Engineering of Reconfigurable Hardware/Software Objects*, A. Arabnia, Ed. June 2000, CSREA Press.

[13] A.C. Guillou, P. Quinton, T. Risset, and D. Massicotte, "High Level Design of Digital Filters in Mobile Communications," DATE Design Contest 2001, Mar. 2001, Second place.

[14] T. Risset D. Cachera and D. Zegaoui, "Formal bit with determination for nested loop programs," Tech. Rep. 1389, Irisa, Rennes, 2001.

[15] D. Cachera, P. Quinton, S. Rajopadhye, and T. Risset, "Proving Properties of Multidimensional Recurrences with Application to Regular Parallel Algorithms," in *6th International Workshop on Formal Methods for Parallel Programming: Theory and Applications (FMPPTA)*, San Francisco, Apr. 2001.

[16] P. Quinton, S. Rajopadhye, and D. Wilde, "The Naive Execution of Affine Recurrence Equations," Tech. Rep. 904, Irisa, Rennes, France, Jan. 1995.

[17] F. Quilleré, S. Rajopadhye, and D. Wilde, "Generation of Efficient Nested Loops from Polyhedra," *International Journal of Parallel Programming*, vol. 28, no. 5, pp. 469–498, 2000.

[18] D. Wilde and S. Rajopadhye, "Memory reuse analysis in the polyhedral model," in *EUROPAR 96: Parallel Processing Conference*, Lyon, France, August 1996, pp. 389–397, Springer Verlag.

[19] Fabien Quilleré and Sanjay Rajopadhye, "Optimizing memory usage in the polyhedral model," *ACM Transactions on Programming Languages and Systems*, vol. 22, no. 5, pp. 773–815, sep 2000.

[20] VHDL Synthesis Interoperability Working Group IEEE PAR 1076.6, "IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis , IEEE P1076.6-1999," 1999, available at http://standards.ieee.org/catalog/design.html.