Synthesizing Neural Nets into Image Processing Hardware

Using the CCC Behavioral Synthesizer

Michael F. Dossis[*1], Dimitrios E. Amanatidis[2]

Department of Informatics and Computer Technology, TEI of Western Macedonia
Kastoria Campus, Fourka Area, Kastoria, GR 52100, Greece
[*1]dossis@kastoria.teikoz.gr; [2]d_amanatidis@yahoo.gr

Abstract- **An integrated, formal high-Level synthesis (HLS) framework is used in this work for hardware implementation of cellular neural networks, which are used in real time image processing. The Custom Coprocessors Compilation (CCC) HLS behavioral synthesiser generates correct-by-construction register transfer – level (RTL) VHDL hardware models of computation-intensive applications. Thus, time-consuming RTL and gate-level simulations are avoided and verification time is cut down to a fraction of the usual time that takes to achieve the same goal with traditional approaches. Such applications include image processing with cellular neural networks (CNNs). The synthesizer utilizes formal compiler-compiler and logic programming techniques, to transform algorithmic ADA into RTL VHDL or Verilog which are directly implementable into hardware using any available RTL synthesizer. The CNNs were rapidly coded, compiled and verified along with all the necessary testbenches in GNU ADA. The applications targeted here are edge-detection, halftoning and morphological processing, which are used to evaluate the CCC HLS framework. The contribution of this work is hardware implementation of CNNs using the CCC HLS tools to formally, and rapidly develop, verify and prototype advanced image processing applications.**

Keywords- Formal Methods; High-Level Synthesis; VHDL; ADA; High-Level Verification; Image Morphological Processing; Cellular Neural Networks

I. INTRODUCTION

Developments in device integration technology helped to realise extremely complex systems into a single integrated circuit (IC) or a small set of ICs that are used in multimedia and consumer devices, medical and scientific data processing, embedded systems, telecommunications, industrial and vehicle electronics and control, banking, aerospace, avionics, naval and transport infrastructure, with more target areas emerging every year. Industry and academia invested in research for rapid and automated methodologies in order to allow the development of these complex computing systems on time [1]. The set of the investigated methodologies was dominated by high-level synthesis (HLS) and electronic system-level (ESL) design, which borrowed technology from software compilers and established E-CAD systems and combined those with new optimizing transformation techniques in order to improve the delivered synthesis results.

The basic HLS optimizations are scheduling, allocation and binding. While obeying in precedence constraints, scheduling attempts to parallelise as many as possible operations into the same control step (state). Allocation determines the kind and number computation, storage, and interconnection units that will implement the operations, variables, and data transfers of the source code, respectively. Finally, variables, constants, operations and data transfers are mapped onto registers, wires, functional units and interconnections (wires and multiplexers), respectively by binding.

This work contributes with a rapid, automated and formal approach to deliver hardware implementations which customised to the specific computational requirements of image processing algorithms. These algorithms are implemented in hardware with Cellular Neural Networks (CNNs). Image processing applications of CNNs include: half-toning, edge and corner detection, connected element detection and morphological processing such as erosion, dilation and hole filling [2]. Because edges are the boundary between various image objects, edge-detection is one of the most important tasks in element detection. Low-level pixel is used for higher-level tasks such as segmentation, object recognition and registration, therefore they utilise edge-detection for these image processing tasks.

Our HLS framework is called Custom Coprocessor Compilation (CCC) framework[1]. CCC automatically transforms arbitrary, high-level, behavioral program code (with complex control flow) into provably-correct hardware implementations. The generated hardware includes all the necessary addressing, communication with local or external memories. In this way, CCC drastically reduces the specification, design, verification, implementation and prototyping cycles by orders of magnitude. The CCC tool consists of the frontend compiler, the intermediate format and the backend compiler. The Intermediate Predicate Format (IPF)[2] is coded with logic predicate clauses which are loaded into the backend compiler. The backend compiler is based on the logic resolution of Horn clauses to implement an inference engine with logic rules and deduction [3].

[1] This hardware synthesis method is patented with patent number: 1005308, 5/10/2006, from the Greek Industrial Property Organization.
[2] The Intermediate Predicate Format is patented with patent number: 1006354, 15/4/2009, from the Greek Industrial Property Organization.

Section II outlines existing work. The CCC method is explained in Section III. Section IV discusses modelling, verification and experimental results and Section V draws conclusions and suggests future extensions of this work.

II. Existing Work and Background Literature

A. Existing Work in High-Level Synthesis

An early HLS approach in [4] transforms algorithmic Digital System Specification Language code into hardware. Systems are accelerated in [1] with the use of Field-Programmable Gate Array (FPGA)-based accelerators. Proprietary input formats, used for particular application targets (e.g. DSP) are analysed in [5], [6] and [7]. HLS scheduling algorithms are reviewed in [4], [8]-[10].

Intermediate translation formats, are reported in [11]–[14]. Interfaces are synthesised and embedded in the core hardware functions, and protocol conversion circuitry is generated for connecting multiple modules with various communication protocols, with the host environment in [15]. A small subset of ANSI C code is optimised, into hardware implementations in [16]. CCC accepts all of the standard programming constructs (e.g. for, while, do loops, nested loops and if-then-else blocks, etc.) and the quality of the produced hardware implementations is superior to that of the available HLS tools.

Data-flow descriptions are optimised in [17] using Taylor Expansion Diagrams. The power consumption of memory elements is reduced, with the use of dual power supply voltages [18]. System C models are synthesized into hardware using the System Co Designer tool [19]. Optimal hardware co-processors are formally and automatically generated in this work, from custom specifications in ADA code.

Compiler-generators are used to automatically create large parts of the CCC synthesizer from formal grammar definitions [11], [13]. The advantage of the CCC framework is that it accepts arbitrary code with complex control flow, and with hierarchical subroutines calling other subroutines. Moreover, all of the required hardware-to-hardware and system-to-hardware interfaces and communication protocols (e.g. with main memory) are automatically generated by the CCC.

B. Cellular Neural Networks

CNNs were introduced by Chua and Yang [20], [21]. In order to process large data in real time, matrices of simple, locally coupled, nonlinear, and dynamic cells are utilised. This architecture originated from the Cellular Automata and the Hopfield neural network structures, which are convenient for hardware implementation. Using these structures, image processing and partial differential equation solving, time-consuming tasks can be executed. CNNs can process information in continuous-time, and they are interconnected locally, which makes them suitable to implement in VLSI hardware (a fully connected CNN implements a typical Hopfield network). In this work, the hardware image processing algorithms are modelled with fully connected CNNs coded in ADA behavioral programs.

A two-dimensional CNN array with M rows and N columns, features cell dynamics, given by the following equations:

$$\dot{x}_{ij}(t) = -x_{ij}(t) + \sum_{kl \in S_{ij}} a_{kl} y_{kl}(t) + \sum_{kl \in S_{ij}} b_{kl} u_{kl} + I_{ij} \quad (1)$$

$$y_{ij} = f(x_{ij}) = (|x_{ij} + 1| - |x_{ij} - 1|)/2 \quad (2)$$

where xij(t) is the cell state, uij is the static input, yij(t) is the cell output, Iij is a constant bias and aij, bij are weighting coefficients. Assuming 8-neighbor connectivity (Sij), the coefficients forming the two 3×3 templates are: feedback template A and control template B (see Fig. 5). Exploring appropriate templates and bias (a string of 19 real numbers called a CNN gene) for each application is an active research activity [22], [23]. The dynamic functions of a CNN are fully defined by the initial state xij(0), static input uij. In [24], there is a report and analysis of a big library of templates.

C. Classification and Development Techniques for CNNs

It is widely accepted that the capabilities of CNNs can be fully exploited using hardware implementations of them. A sample of this argument is discussed in this work. Because hardware is becoming cheaper, parallel and fast processing is the dominant reason for custom hardware implementation of CNNs. The contribution of this work is the formal and rapid modelling and prototyping of CNNs, targeting any FPGA and ASIC (Application-Specific Integrated Circuit) technologies, because the generated VHDL or Verilog RTL models are provably-correct and technology/tool vendor - independent.

Existing systems are implemented with a wide range of neural network techniques. A classification in [25] includes neuro-chips or neuro-computers, and general and special purpose architectures, categorised by the number of neurons, the neuron state (digital/analog), the bit precision, the number of synapses, the weights, the activation (probabilistic/deterministic), learning (on chip / off chip), speed (learning/processing), ability to cascade, the implementation technology, the number and type of I/Os, and the clock and data transfer speed [26]. A snapshot of this classification is shown in Fig. 1.

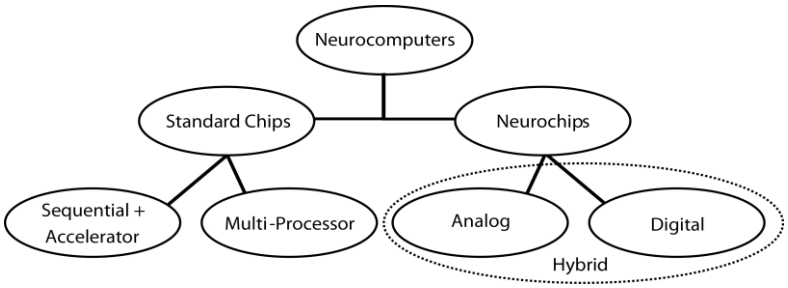


Fig. 1 Categories of Neural Network hardware

Analog, digital or hybrid ASICs can be used to implement neural nets. Analog CNNs became popular, when a combination of analog spatial/temporal dynamics and logic architecture was invented, namely the CNNUM [27] architecture. This leads to the emergence of a number of analogic processors. A fully programmable 128x128 array processor, designed in Seville [28] and implemented by STMicroelectronics in Catania [29], is the most advanced analog CNN chip. The previous generation (64x64) chip, is known as ACE4k. This chip was used in the ALLADIN network [30], in a combination with a DSP processor, and implementation of the first high performance, industrial quality, visual computer, which used 4096 parallel cell processors, with a processing rate of up to 10000 frames/second (64x64 images) [31].

FPGAs can be used instead of ASICs to implement popular neural networks [32]. FPGAs can implement "soft" circuitry, for data processing without the overhead of a microprocessor and operating system. Neural networks are also highly scalable, and therefore they can be widely used in ever upgrading FPGA system implementations. FPGAs can also be totally reconfigured on line by software download or via the internet to apply extensions, bug fixes, new releases or total change of functionality throughout the development and lifetime of the product. This constitutes an advantage and contribution of this work, since rapid re-development and re-configurability are necessary for such devices, so as to support new standards and applications. Popular and widely-available CAD tools are used to configure FPGAs. Designs are modelled with Hardware description languages (HDLs) like VHDL, Verilog or ABEL. In [33] FPGA implementation of a CNN using Handel-C, is discussed. The ADA programming language is used here, a highly-reliable high-level language, to code hardware accelerators. In this way, executable formal specifications are used, since ADA is used for safety-critical and high-reliability applications.

III. MODELING AND IMPLEMENTATION WITH THE CCC TOOLS

The HLS CCC development flow is outlined in Fig. 2. The behavioral ADA model of the design is first modelled and verified in pure algorithmic ADA code. The frontend compiler translates this model into the IPF (Intermediate Predicate Format) database, using traditional but formal compiler techniques. Compiler-generator from formal grammar is used, so the translation is correct-by-construction. IPF captures the information of the ADA programs in a number of Prolog clauses (facts). The backend compiler is a logic inference engine that optimizes the IPF clauses [3] into synthesizable RTL hardware models. The generated VHDL/Verilog capture the finite state machine (FSM) and the data path of the hardware. Thus, the CCC hardware compilation generates provably-correct implementations.

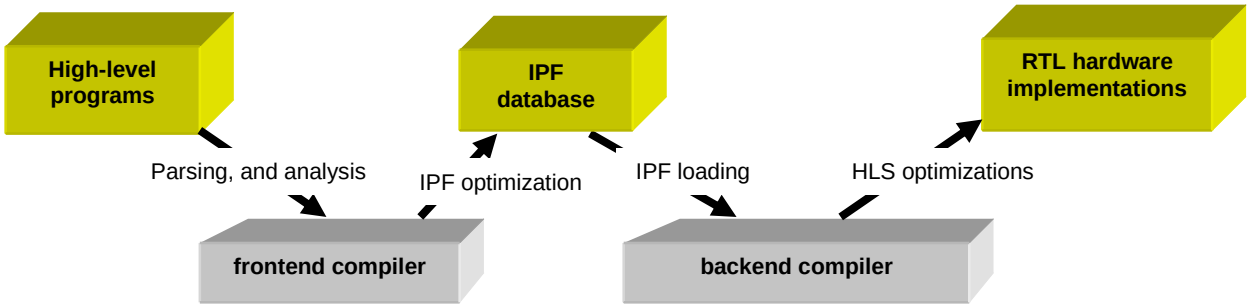


Fig. 2 The CCC, automated & formal, hardware synthesis flow

The formal, logic programming rules that implement the backend compiler inference engine look like the predicate clause:

$$A_0 \leftarrow A_1 \wedge \ldots \wedge A_n \text{ (where } n \geq 0) \qquad \text{(form 1)}$$

where $\leftarrow$ is the logical implication symbol ($A \leftarrow B$ means that if B applies then A applies), and A_0, …., A_n are atomic formulas (logic facts) of the form:

$$\text{predicate\_symbol(Var\_1,…, Var\_N)} \qquad \text{(form 2)}$$

where the positional parameters Var_1,…, Var_N of the above predicate_symbol 错误!未找到引用源。 are either variable names (in the case of the backend compiler logic rules), or constants (in the case of the IPF table statements). Predicate form 2 is an example of the IPF facts. IPF includes these logic facts, grouped in the IPF tables. Each IPF table contains a list of

homogeneous facts, e.g. all prog_stmt facts for a given subprogram are grouped together in the program statements table for a certain ADA module.

The combination of frontend and backend compilers formally transforms the input ADA subroutines in an equivalent number (using a 1-to-1 compilation) of provably-correct hardware description language (HDL) modules. This is achieved with the optimising transformation predicates of the backend inference engine. In this way, rapid and formal hardware prototyping is achieved on our CNN image processing architectures.

An optimised operation scheduler called Parallel-Abstract Resource – Constrained parallelizing scheduler (PARCS) is one of the major optimising transformations of the backend compiler. The basic steps of the PARCS algorithm which is coded in Prolog, are listed as pseudo-code in **错误!未找到引用源。**. The code shows that both complex datapath and control flow operations can be processed by the PARCS.

1. Begin with the initial schedule (including the custom and special external memory operations)
2. Set current PARCS state to 1
3. Get the 1st initial schedule state and make the 1st the current state
4. Get the next initial schedule state
5. Find out whether the next state's operations have any dependencies with the current state
6. If no dependencies, then absorb the next state's operations into the current PARCS state; If there are dependencies then conclude the so far absorbed operations into the current PARCS state, store the current PARCS state, PARCS state <- PARCS state + 1
7. Make next PARCS state the current state; store the new state's operations into the (new) current PARCS state
8. If next initial schedule state = conditional then call the conditional (true/false branch) processing predicates, else continue
9. If there are more states to process then go to Step 4, otherwise conclude the so far operations of the current PARCS state and terminate

Fig. 3 The PARCS scheduler pseudo code

PARCS attempts to parallelise as many operations as possible in the same FSM control state, while checking to satisfy data and control flow dependencies. Large data structures can be optionally located on the shared memory using a set of memory options stating their position, size and interface ports. In this case, all of the necessary communication protocols and addressing functions are generated automatically and co-optimized with the rest of the module's operations. Thus, overall performance is improved. A part of the edge-detection ADA program which models the CNN core is shown in Fig. 4.

```
1.    --Euler and convolution main loop
2.    Tmax := 10;
3.    FOR T IN 1..Tmax LOOP
4.      FOR J IN 0..HEIGHT-1 LOOP
5.        FOR I IN 1..WIDTH LOOP
6.          Cnn_Convolute(I,J,A,B,Y,U,Sa,Sb);
7.          --reshape by rows
8.          X(J*WIDTH+I) := Sa+Sb+Bias;
9.          Temp := X(J*WIDTH+I);
10.         Y(J*WIDTH+I) := Cnn_Pwl(Temp);
11.       END LOOP;
12.     END LOOP;
13.   END LOOP;
```

Fig. 4 The core of CNN edge-detection algorithm in ADA

This code part underlines the expressive power of our ADA approach. If contains three level nested for loops and complex conditional operations, emphasizing the CCC capability to model both dataflow and control flow – intensive designs. The CCC transformation of this algorithm took in less than 5 minutes. Modeling and verification of this algorithm took less than a couple of hours using the GNU ADA development tools.

IV. Algorithm Verification and Experiments

We modelled the CNN tasks in ADA. These tasks process the grey-scale or binary images (initial state and input) and they read the task-specific templates (A, B and Bias). The matrices for edge detection are shown in Fig. 5. We use the .ppm (portable pixel maps) format to accept color images. Fig. 6 shows the edge detection results after ten iterations on "Lena".

A =

0	0	0
0	2	0
0	0	0

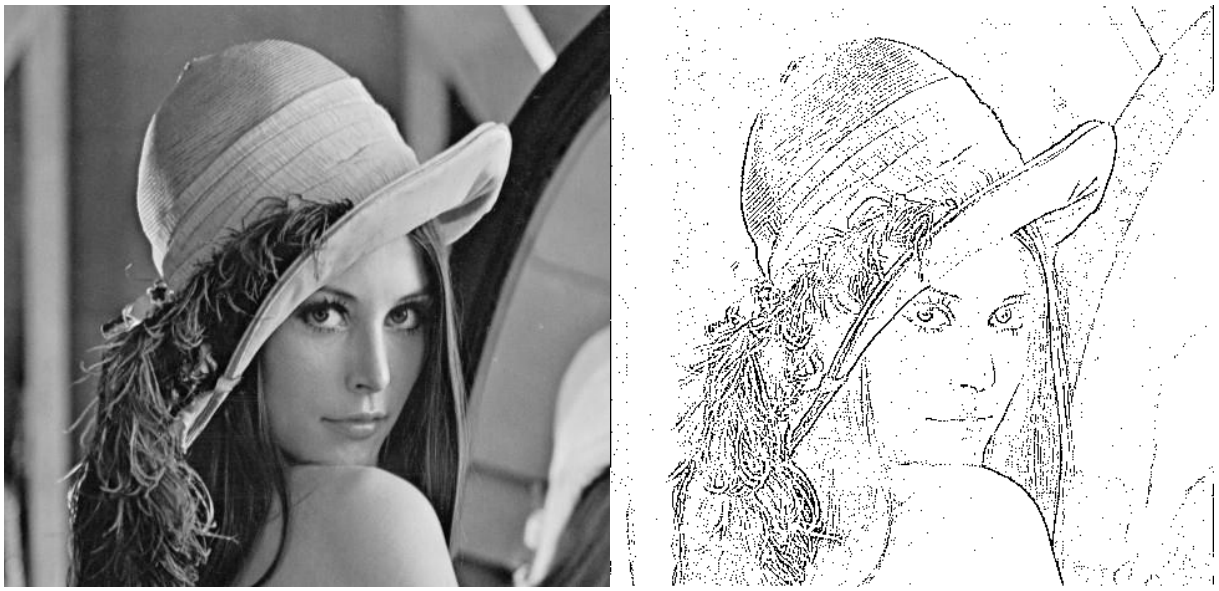
B =

-1	-1	-1
-1	8	-1
-1	-1	-1

Bias = -191

Initial State X(0) = white (255) Input U = original image

Fig. 5 Edge detection, CNN template matrices



(a) Original "Lena" image (b) Edge detection output after 10 iterations

Fig. 6 CNN edge detection

By modifying the templates and the initial state of CNNs, we can upgrade their processing on the input data. Fig. 7 shows the result of halftoning and Fig. 8 shows the CNN result of image morphology-related tasks such as dilation and erosion. Fig. 6(a) is the same input image which is passed onto the CNN algorithms for processing. Template B (Fig. 5) is used for the structuring element of morphological image processing tasks. Dilation followed by erosion is also called image closing, while the reverse order is called image opening. Our opening algorithms smooth image contours, break narrow isthmuses and eliminate thin protrusions. Our closing algorithms narrow smooth sections of contours, merge narrow breaks and long thin gulfs, eliminate small holes, and fill gaps in contours [34]. All of these tasks were modelled and verified in ADA, in a way that it is indicated in Fig. 4. Then the CCC HLS toolset was used to automatically convert these tasks into synthesizable and simulatable RTL VHDL implementations. For the sake of evaluating the HLS tool, RTL simulations were executed. Up to here!

Fig. 7 CNN halftoning

(a) dilation (b) erosion

Fig. 8 CNN image closing

Fig. 8 shows the result images after dilation and erosion, which were produced in this order (closing) by successive CNN operations on the edge image.

ADA testbenches were built to contain function calls to our CNN ADA task libraries (packages in ADA's terminology). Thus, all these tasks were rapidly verified in ADA using fast compile-and-execute. The tests generated the test vectors for the golden model. The generated RTL (VHDL) was instantiated in VHDL testbenches that were simulated using the Modelsim simulator from Mentor Graphics, to verify the correctness of the CCC translation. Test vectors (initial state and input images) were fed into the CNN algorithm (Model Under Test or MUT) as shown in Fig. 9. RTL simulations were compared to the ADA Golden Model verification, to indicate verification pass or fail (Fig. 9). The CNN core ADA function was transformed to RTL using our HLS compiler, with experiments for massively parallel or conventional FSM + datapath architectures, and with the large arrays residing on target system main memory. The memory read/write ports and communication protocol command sequences, for the external shared memory, were also automatically generated and optimized.

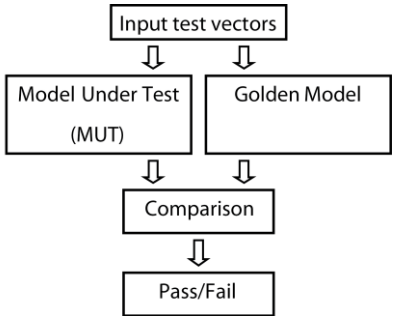


Fig. 9 High-level, system verification flow

Using the PARCS optimizing scheduler, the image erosion, dilation and half-toning tasks were optimized from 63 (initial schedule) states down to 41 (optimized schedule) states, a reduction of almost 35%. PARCS reduced the edge-detection schedule from 105 (initial schedule) down to 74 states (optimized schedule), a reduction of about 30%. These statistics are shown graphically in Fig. 10. The compilation took less than 3 seconds on a conventional Intel Centrino laptop PC.

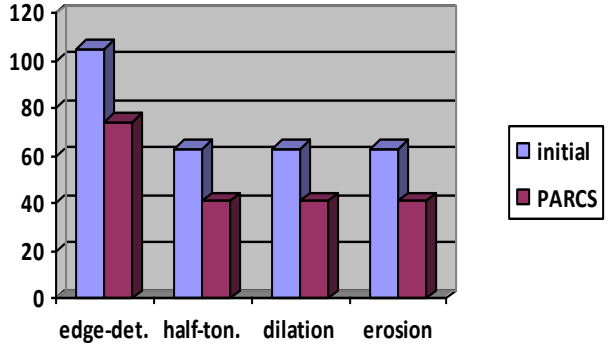


Fig. 10 Initial and optimized (by PARCS) schedule state statistics

With the completion of ADA verification and CCC HLS synthesis, RTL synthesis was executed on the edge-detection implementation, and then the designs were implemented on a Xilinx Virtex-4 FPGA, with a number of commercial synthesizers. These experiments produced implementations featuring clock rates up to 104 MHz. The implementation used 2 DSP48 FUs, 1144 FFs and 343 multiplexers for the FSM+datapath option vs 8 DSP48s, 1257 FFs and 250 multiplexers for the massively parallel option.

V. Conclusion

The contribution of this work is very significant since CNNs constitute generic, powerful, scalable and versatile image processors. Moreover, CNNs are suitable for ASIC/FPGA implementations. By rapidly transforming high-level CNN models into RTL implementations, we proved that the CCC HLS compiler contribution is invaluable; behavioral synthesis was automatic, of high-quality, very fast and the delivered results are correct-by-construction. All CNNs for edge-detection, halftoning, dilation and erosion, were modeled in ADA, transformed into VHDL with the CCC tools, and verified both in ADA and VHDL testbenches (to prove the principle of formal transformation). This rapid and formal nature of the CCC compilation, manifests the contribution of this work. The produced RTL code is generic and independent of any specific vendor or hardware template, therefore our synthesis flow can be connected to any existing design and product development flow. The method of this work reduced some months of development effort into a few hours. Because generic and common program constructs are accepted, it is possible to extend our modelling approach to other programming languages (such as ANSI-C). In any cases, the user is freed from complex hardware details such as control states, interfaces and hardware and architecture structures.

Future work in this area includes: a more detailed extension to this framework and experimentation with other neural and non-neural image processing algorithms such as PCNN (Pulse-Coupled Neural Networks) or optical flow. Also, there is ongoing work for extending the frontends of the CCC framework with more input forms such as ANSI-C, UML, Matlab, etc. Moreover, the backend HDL writers will be extended to generate other formalisms such as Verilog HDL, and System-C. Ongoing work will soon deliver capabilities for IP support, multi-cycle and single-cycle custom arithmetic functions and hardware macro blocks, such as fast multipliers and floating point processing units.

References

[1] R. Pellizzoni and M. Caccamo, "Real-Time Management of Hardware and Software Tasks for FPGA-based Embedded Systems," *IEEE Transactions on Computers*, vol. 56, pp. 1666–1680, 2007.

[2] T. Yang, Handbook of CNN Image Processing: *All you need to know about Cellular Neural Networks*. Yang's Scientific Research Institute, 2002.

[3] U. Nilsson and J. Maluszynski, *Logic, Programming and Prolog*. John Wiley & Sons Ltd., 2nd Edition, 1995.

[4] R. Camposano and W. Rosenstiel, "Synthesizing circuits from behavioural descriptions," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 8, no. 2, pp. 171–180, 1989.

[5] A. Casavant, M. d'Abreu, M. Dragomirecky, D. Duff, J. Jasica, M. Hartman, K. Hwang, and W. Smit, "A Synthesis Environment for Designing DSP Systems," *IEEE Des. Test*, vol. 6, no. 2, pp. 35–44, March 1989.

[6] I. Auge´, F. Pe´trot, F. Donnet, and P. Gomez, "Platform- based design from parallel C specifications," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 24, no. 12, pp. 1811–1826, 2005.

[7] M. C. Molina, R. Ruiz-Sautua, J. M. Mendias, and R. Hermida, "Bitwise scheduling to balance the computational cost of behavioral specifications," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 25, no. 1, pp. 31–46, 2006.

[8] R. A. Walker and S. Chaudhuri, "Introduction to the Scheduling Problem," *IEEE Design & Test of Computers*, vol. 12, no. 2, pp. 60–69, 1995.

[9] A. A. Kountouris and C. Wolinski, "Efficient scheduling of conditional behaviors for high-level synthesis," *ACM Trans. Design Autom. Electr. Syst*, vol. 7, no. 3, pp. 380–412, 2002.

[10] P. G. Paulin and J. P. Knight, "Force-Directed Scheduling for the Behavioral Synthesis of ASIC's," *IEEE Transactions on CAD*, vol. 8, no. 6, pp. 661–678, Jun. 1989.

[11] A. V. Aho and J. D. Ullman, *Principles of Compiler Design*. Addison-Wesley, 1978.

[12] J.-P. Tremblay and P. G. Sorenson, *The Theory and Practice of Compiler Writing*. McGraw-Hill Inc., 1984.

[13] W. Waite and G. Goos, *Compiler Construction*. New York, USA: Springer-Verlag Inc, 1984.

[14] R. Hunter, *Compilers their design and construction using Pascal*. Chichester, England: John Wiley & Sons Ltd, 1985.

[15] B. Lin and S. Vercauteren, "Synthesis of concurrent system interface modules with automatic protocol conversion gen- eration," in *Proc. International Conference on Computer-Aided Design (ICCAD'94)*. Los Alamitos, Ca., USA: IEEE Computer Society Press, Nov. 1994, pp. 101–109.

[16] S. Gupta, R. K. Gupta, N. D. Dutt, and A. Nicolau, "Coordinated parallelizing compiler optimizations and high-level synthesis," *ACM Trans. Design Autom. Electr. Syst*, vol. 9, no. 4, pp. 441–470, 2004.

[17] D. Gomez-Prado, Q. Ren, M. J. Ciesielski, J. Guillot, and E. Boutillon, "Optimizing data flow graphs to minimize hardware implementation," *in Proc. of DATE*. IEEE, 2009, pp. 117–122.

[18] I. Shin, S. Paik, and Y. Shin, "Register allocation for high- level synthesis using dual supply voltages," *in Proc. DAC. ACM*, 2009, pp. 937–942.

[19] C. Haubelt, T. Schlichter, J. Keinert, and M. Meredith, "SystemCoDesigner: automatic design space exploration and rapid prototyping from behavioral models," *in Proc. of the 45th Design Automation Conference, DAC 2008*, Anaheim, CA, USA, June 8-13, 2008, L. Fix, Ed. ACM, 2008, pp. 580–585.

[20] L. O. Chua and L. Yang, "Cellular Neural Networks: Theory," *IEEE Transactions on Circuits and Systems*, vol. 35, no. 10, pp. 1257–1272, 1988.
[21] L. O. Chua and L. Yang, "Cellular Neural Networks: Applications," *IEEE Transactions on Circuits and Systems*, vol. 35, no. 10, pp. 1273–1290, 1988.
[22] A. Zarandy, "The art of CNN template design," *International Journal of Circuit Theory and Applications*, vol. 27, no. 1, pp. 5–23, 1999.
[23] M. Hanggi and G. Moschytz, *Cellular Neural Networks Analysis, Design and Optimization*. Kluwer Academic Publishers, 2000.
[24] K. Karacs, Gy. Cserey, A. Zarandy, P. Szolgay, Cs. Rekeczky, L. Kek, V. Szabo, G. Pazienza and T. Roska, "Software Library for Cellular Wave Computing Engines in an era of kilo-processor chips, Version 3.1," Cellular Sensory and Wave Computing Laboratory of the Computer and Automation Research Inst., Hungarian Academy of Sciences and the Jedlik Laboratories of the Pazmany P. Catholic University, Tech. Rep., 2010.
[25] I. Aybay, S. Cetinkaya, and U. Halici, "Classification of Neural Network Hardware," *Neural Network World IDG Co*, vol. 6, no. 1, pp. 11–29, 1996.
[26] J. Heemskerk, "Overview of Neural Hardware. In: Neuro-computers for Brain-Style Processing. Design, Implementation and Application," Ph.D. dissertation, Unit of Experimental and Theoretical Psychology, Leiden University, The Netherlands, 1995.
[27] L. Chua and T. Roska, "The CNN Universal Machine: An Analogic Array Computer," *IEEE Transactions on Circuits and Systems-II: Analog and Digital Signal Processing*, vol. 40, no. 3, pp. 163–173, 1993.
[28] http://www.imse.cnm.es/, Institute of Microelectronics at Seville.
[29] http://www.st.com/, Catania STMicroelectronics.
[30] http://lab.analogic.sztaki.hu/, Analogic and Neural Computing Systems Laboratory.
[31] L. Fortuna, P. Arena, D. Balya, and A. Zarandy, "Cellular Neural Networks: A Paradigm for Nonlinear Spatio-Temporal Processing," *IEEE Circuits and Systems magazine*, vol. 1, no. 4, pp. 6–21, 2001.
[32] A. R. Omondi and J. C. Rajapakse, Eds., *FPGA Implementations of Neural Networks*. Springer Netherlands, 2006.
[33] D. Amanatidis, D. Tsaptsinos, P. Giaccone, and G. Jones, "Image Processing using CNNs and FPGAs: Initial Results," *in Proc. of the WSEAS International Conference on Neural Networks and Applications (NNA)*, Puerto de La Cruz, Tenerife, Canary Islands, Spain, Feb. 2001, pp. 4651–4655.
[34] R. C. Gonzalez and R. E. Woods, *Digital Image Processing*, Third Edition. Prentice Hall, 2008.

Michael F. Dossis has an advanced Engineering Diploma (MSc-equivalent) from 5-year course in Electrical Engineering from National Technical University of Athens, Greece (1990) and a PhD in Electronic and Electrical Engineering in the field of High-Level Synthesis from University of Bradford, UK (1994).

He has a long industrial engineering experience, mainly as a system and ASIC engineer in companies such as LSI Logic, ARM Ltd, Virata (now Connexant) and Intracom Telecom S.A. He has also post-doc research experience in European Projects in Bradford University, the University of Oxford, UK and Intracom Telecom, Greece. He has published in scientific journals and conferences, in the areas of HLS, compilers, E-DA, electronic CAD tools, VLSI, ASIC and FPGA design. His research interests include HLS, computer languages and their compilers, embedded systems, digital system design methods, formal methods and ESL for E-DA, and computer programming.

Dr. Dossis is an Associate Professor in Informatics and Computer Technology of the Technical Educational Institute of Western Macedonia, Greece. He is a member of the Technical Chamber of Greece (TEE) and reviewer and organizing committee member of a number of scientific journals and conferences.

Dimitris E. Amanatidis graduated in 1993 with a Diploma in Mathematics from University of Crete in Greece and in 1997 with a M.Sc. in Computer Science (Distributed Systems) from University of Essex, UK. On October 2008 he was awarded a PhD ("Motion Estimation and Segmentation of Colour Image Sequences") from the Faculty of Computing, Information Systems and Mathematics at Kingston University.

His professional experience includes almost 15 years of teaching at various academic and research institutions in Greece and UK, participation in a number of EU-funded projects and employment as a researcher for Chemical Process Engineering Research Institute in Greece. Research interests and related publications are mainly on the fields of image processing and pattern recognition, neural networks and optimization, parallel programming and FPGAs.

Dr. Amanatidis is currently an Adjunct Lecturer in Informatics and Computer Technology Department of the Technical Educational Institute of Western Macedonia, Greece and in Technology Management Department of the University of Macedonia. He is also involved in reviewing for scientific journals.