LEARNING HARDWARE USING MULTIPLE-VALUED LOGIC

Part 1: Introduction and Approach

THE AUTHORS PROPOSE A LEARNING-HARDWARE APPROACH AS A GENERALIZATION OF EVOLVABLE HARDWARE. A MASSIVELY PARALLEL, RECONFIGURABLE PROCESSOR SPEEDS UP LOGIC OPERATORS PERFORMED IN LEARNING HARDWARE. THE APPROACH USES COMBINATORIAL SYNTHESIS METHODS DEVELOPED WITHIN THE FRAMEWORK OF THE LOGIC SYNTHESIS IN DIGITAL-CIRCUIT-DESIGN AUTOMATION.

Marek Perkowski
David Foote
Qihong Chen
Anas Al-Rabadi
Portland State University

Lech Jozwiak
Eindhoven University of Technology

Many robotics, multimedia, and other types of tasks require fast, real-time realization of certain algorithms to solve problems presented by a human's or robot's environment. This need is most common in computer vision and speech recognition areas, where specialized hardware accelerators are built to satisfy speed, power, size, or cost requirements. So far, most of these hardware systems solve polynomial problems. However, in test generation[1] and combinatorial optimization areas, reconfigurable, field-programmable-gate-array (FPGA)-based processors can solve some nondeterministic polynomial time (NP)-hard problems, such as satisfiability,[2] tautology or binate covering,[3] and so on.

Hanyu et al. describe a robot vision coprocessor that performs matching by solving the NP-hard maximum-clique problem.[4] Software solutions are adequate for many other NP-hard problems in computer vision, robot navigation and manipulation, speech recognition, and so on. Although these software solutions are too slow to be applied in real time, require too many parallel processors to put on a robot, or are too approximate to produce useful solutions. Building intelligent robots will ultimately require developing real-time hardware realizations of exact or high-quality approximate algorithms for NP combinatorial problems. We propose a new approach that combines hardware realization of such algorithms with learning in reconfigurable hardware.

Observe recent rapid progress in soft computing, that is, artificial neural networks (ANNs), fuzzy logic, rough sets, genetic algorithms, and genetic and evolutionary programming. In different ways, these approaches try to solve complex and poorly defined problems that previously developed analytic models could not efficiently tackle. All of these approaches offer a method of automatic learning. These methods teach the computer system by examples and evaluations of the system's behavior rather than completely programming the system. This philosophy also dominates approaches to artificial life, problem solving using analogies to nature, decision making, knowledge acquisition, and intelligent robotics. Machine learning has become a general paradigm for software system design, unifying all these previously disconnected areas. With the

0272-1732/02/$17.00 © 2002 IEEE

invention of evolvable hardware,[5-7] machine learning becomes a new hardware construction paradigm as well. Several circuit applications evolved directly in hardware and were realized in reconfigurable FPGAs.[6,8]

The term *evolvable hardware* originally designated the realization of genetic algorithms in reconfigurable hardware but now extends to other evolutionary algorithms such as genetic programming. The evolvable-hardware approach to computing has raised considerable interest and enthusiasm among some researchers, but it has also raised severe skepticism among others. Currently, the evolvable-hardware approach dominates learning in reconfigurable hardware (with some exceptions.[9,10]).

While agreeing with the importance of learning algorithms realized in hardware and particularly in reconfigurable hardware, we might ask: Why use evolutionary algorithms? Practical experiences in the past 10 years prompt us to question the usefulness of evolutionary algorithms as a sole learning method to reconfigure binary FPGAs. Instead, we propose the learning-hardware paradigm,[10] which uses feedback from the environment (for instance, positive and negative examples from the human supervisor) to create a classical, sequential, binary network, directly realizable in standard FPGAs. Our universal logic machine approach[11-14] proposes developing learning machines based on logic principles—in particular, temporal logic,[10] constructive induction,[15-18] and rough-set theory.[19] Algorithms for the previously discussed application areas require fast operations on complex logic expressions and solving NP problems such as satisfiability, graph coloring, or set covering. These algorithms should therefore be realized directly in hardware to obtain the speedups necessary for the real-time operation execution. We present a subset of the learning-hardware model that reduces the learning process to the automatic induction of a combinational network from a set of examples.

The genetic algorithm is a simple and noninformed problem-solving mechanism. Its main operations of crossover and chromosome mutation are easily realizable in hardware using registers that represent chromosomes and some partially random register-transfer operations on the contents of these registers. Hardware implementation of the fitness function computation that evaluates the offspring from crossover and mutation can be very difficult.

Theoretical analysis and our past experience suggest that evolutionary mechanisms alone cannot efficiently produce sufficiently good results for problems of interest. This is especially true within the constraints of real-time operation; the algorithms are often not convergent for real-life data. In contrast, logic-based algorithms that draw upon human knowledge are near optimal and mathematically sophisticated. They lead to high-quality learning results that have no overfitting and small learning errors, or those that lead to knowledge generalization or discovery.[20-26] However, the software realizations of these algorithms use complex data structures (such as binary decision diagrams[26]) and controls that are difficult to directly realize in hardware. For instance, recursion and decision diagrams are difficult to parallelize or pipeline and, in general, to implement in hardware structures. Thus, implementations should take a different approach to the speedup of these algorithms.

In this article, when referring to *learning hardware*, we define the term *learning* as any mechanism that leads to an improvement of operation. Such a definition therefore includes evolution-based learning as a special case of learning hardware. Although specific learning concepts and their formalisms differ from one learning approach to another, each approach creates a network design from examples. The algorithm constructs this network—which might be combinational or sequential; binary, fuzzy, continuous, or multiple valued; synchronous or asynchronous—to represent, for future use, the knowledge acquired in the learning phase. The learned-knowledge network then performs computations on old and new data. These computations are very fast because they only evaluate the network on some of its input values, which is done combinationally, that is, without involving memory. Network responses can be correct or erroneous. The algorithm then evaluates the network's behavior again—using some fitness (cost) function—and the learning and running phases alternate. This model is very general and covers neural networks, cellular automata, binary logic used in computational learning theory, and several other learning approaches.[17,18,27-29]

Problem solving

The problem-solving process in the learning-hardware model consists of learning, construction and tuning of a knowledge network, and execution, which uses the learned-knowledge method for computations. These phases are either implemented by separate physical units or as various virtual hardware processes run on the same physical, time-multiplexed, reconfigurable processor. The execution phase uses knowledge, which means that the constructed network processes some new examples (also called objects, minterms, or input vectors). Compared to designing, building, and using a PC, the learning phase is similar to the entire process of conceptualizing, designing, and optimizing PC hardware and software, and the execution phase to using this computer to perform calculations.

It is not likely that the evolutionary methods will be used to evolve the entire PCs in the near future; large teams of highly qualified humans now design them using sophisticated algorithms, including combinatorial algorithms for logic and physical design. If so, why use the evolutionary hardware approach in robot learning or speech recognition systems? The combination of the evolutionary and mathematical approaches can provide effective and efficient solutions to problems or tasks in robotics and the other previously mentioned areas. Our learning hardware is the first attempt to combine evolutionary and mathematical approaches. It integrates the strength of learning and the sophisticated human-developed combinatorial optimization/decision algorithms. The user cannot redesign a standard computer's processor chip. But the learning-hardware approach redesigns its hardware execution part automatically, using new examples provided to the learning-hardware unit.

Logic-based learning

Logic synthesis researchers and engineers in design automation for digital circuits develop efficient logic network synthesis algorithms. For example, they might need to synthesize a disjunctive normal form (DNF) with the minimum number of terms from a set of examples. Researchers in the constructive induction approach to machine learning also independently develop similar algorithms. Michie makes a distinction between black-box and knowledge-oriented concept-learning systems by introducing concepts of weak and strong learning criteria.[30] The system satisfies a weak criterion if it uses sample data to generate an updated basis for improved performance on the subsequent data. It satisfies a strong criterion when the system communicates concepts learned by it in a symbolic form.[18]

Observe that ANNs, fuzzy logic, genetic algorithms, or similar approaches satisfy only the weak criterion, while the logic approach satisfies the strong criterion. To understand why this is so, consider that humans can understand synthesized logical formulas—be they a rule, a Prolog program, or a logic network—which are all symbolic representations, as required to satisfy a strong criterion.

The results of the learning process, and even of the process itself, should be rational. Learning in hardware should include approaches similar to those for teaching humans—that is, those based on symbolic logic rather than the (nearly random) methods that emulate nature. Human thinking involves the abstract use of symbols, rather than assigning numeric weights to neurons or randomly connecting them. So our learning-hardware approach operates at higher and more natural symbolic-representation levels. The built-in mathematical optimization techniques (such as graph coloring or satisfiability) support the principle of Occam's Razor, offering solutions that are provably good in the sense of computational learning theory.[20,25] Moreover, it could be dangerous to have a robot that learns from examples and cannot logically analyze what it has learned; the user would not be able to predict the robot's behavior. Thus, learning at a symbolic (symbol manipulation) level is the first main point of our approach.

In past research, we have used and compared (using software) various network structures for learning: two-level AND/OR (sum-of-products or DNFs),[31] decision trees (using C4.5 software and multilevel decomposition structures.[21-23,32,33] We have also studied various logic, nonlogic, and mixed optimization methods, for example, search,[13] rule-based, set-covering, maximum clique, graph coloring, genetic algorithm (including mixtures of logic and genetic algorithm approaches),[34,35] genetic programming,[36] ANNs, and simulated annealing. We compared the resulting complexity of

our networks (using an approach based on Occam's Razor), as well as various ways of controlling the number of errors in the learning process.[21-24] Because of their strong theoretically proven properties, the decomposed function cardinality and its extensions for multiple-valued (MV) logic[20-22,33] work well as common measures of network complexity.[22,25]

Our conclusion, based on these investigations, is that logic approaches (especially the MV-decomposition techniques) combined with smart heuristic strategies and good data representations usually provide superior results compared to other approaches. This is because of their lower network complexity and fewer learning errors. Other researchers have made these types of comparisons and reached similar conclusions.

In our experiments, evolutionary approaches provided especially poor results.[6,21,35,36] Genetic algorithms might perform well in other applications (such as analog circuit design or specific, isolated subproblems in digital circuit design). But our experience and the literature did not identify a single problem in which a genetic algorithm-based approach was superior to using a handcrafted heuristic algorithm for the design of complete binary or MV-logic networks. This is perhaps the result of the vast experience researchers have in creating efficient logic-synthesis algorithms (for example, more papers have been written on DNF minimization than perhaps on any other engineering topic). Our approach uses this accumulated human experience, rather than reinventing efficient circuits and design algorithms using the evolutionary methods of a standard evolvable-hardware approach.[5-8]

Learning-hardware approach

Developers of evolvable and learning systems have found that the learning and/or the execution phases realized with current software—or even parallel-programming technologies—are too slow for real-life problems, and especially real-time problems. The situation is essentially the same regardless of whether the developer uses an exhaustive combinatorial search, simulated annealing, or evolutionary algorithms that involve millions of populations. Thus, some researchers proposed to speedup parts of the learning and/or execution phases by migrating them from software to hardware. Researchers have proposed many ambitious projects based on ANNs, cellular logic, DNA, simulated evolution, and biologically motivated hardware that remain quite impractical today. These approaches might be successful in the future, when realized on molecular or quantum levels.

Most approaches to evolvable hardware use binary FPGAs, because they are the only mass-scale reconfigurable (reprogrammable) hardware technology currently available. FPGAs are also relatively inexpensive and widely available. (Although researchers are developing evolvable hardware for field-programmable analog arrays, they are still quite immature.) In binary FPGAs, implementations realize all system functionality at the level of binary-logic lookup tables, flip-flops, and memory cells; the learning process should occur at this level as well. Performing the learning on a lower level of individual layout switches, but evaluating the designs on a higher level of neural networks, leads to gross inefficiencies caused by several levels of interpretation—from neural network to cellular automata and finally to FPGA binary switches.[5-6] Evolvable-hardware researchers have reported these inefficiencies. Thus, learning on the level of logic gates and flip-flops is the second main point of our approach.

In our opinion, the learning level of sequential logic nets is more natural than that at the higher level of ANN or fuzzy-logic function arithmetic operations or the lower level of routing FPGA connection paths. First, the network of logic subfunctions represents, in a natural way, the network of knowledge (logic) concepts; this ensures learning stability and enables human explanation. Second, once researchers realize a network using the logic-level resources in FPGA, they should apply efficient logic-design algorithms and realize them in hardware for speedup.

All high-quality design methods in the VLSI/FPGA design area and, especially, powerful electronic design automation (EDA) tools, can and should be used to go from examples to final FPGA chip programming. This is certainly preferable to duplicating them using low-quality, low-level evolutionary algorithms that attempt to combine, in a naive way, high-level logic and physical design phases. Researchers and engineers have spent many years developing EDA tools, including those

for state machines and combinational logic synthesis, technology mapping, placement and routing, partitioning, timing analysis and optimization, and so on. Using these tools to build learning hardware will greatly facilitate the field's goals. Researchers should also use the principle of Occam's Razor; applying it in one form or another makes possible meaningful discoveries and explainable results. Although Occam's Razor is the basis of computational learning theory, it has been largely ignored by evolvable-hardware researchers.

We do not believe that the purist strategies of evolutionary hardware are acceptable for any practical digital design application. Therefore, we propose the concept of learning hardware based on human problem-solving experience and the application of mathematical algorithms, problem-solving strategies, and existing high-quality design methods and tools, rather than on ANNs and genetic algorithms (today's two basic evolvable-hardware methods). Evolution can remain one of the main principles of building next-generation hardware, but researchers should restrict its use to higher abstraction levels, rather than to the lowest level of FPGA resources, or to solving some subproblems. Design tools should also help evaluate and select the most promising design variants before mapping to low-level field-programmable resources. Using genetic algorithms on the switch level leads to very long chromosomes and nonconvergent learning processes.

On the other hand, the idea of realizing probabilistic methods in hardware is promising. The incorporation of probabilistic methods to design automation tools leads to synergistic effects when applied with logic optimization methods that account for some types of constraints, or when combined with deterministic use of some available information.[37]

Ashenhurst decomposition is one general synthesis method for multilevel logic, both binary and MV. This basic decomposition partitions functionality into predecessor and successor blocks. One wire connects the two blocks. The method partitions the original function's input variables into two sets: a bound set that must go to the predecessor block and a free set that goes directly to the successor block. This decomposition works on smaller and smaller blocks until one FPGA-configurable logic block (CLB), standard cell, or if-then-else decision rule realizes each block. In *nondisjoint decomposition*, free and bound sets overlap; a *disjoint decomposition* has no overlap. Every multivalued function or relation is nondisjointly decomposable.[32] The Curtis decomposition is a generalization of Ashenhurst decomposition in which more than one wire connects predecessor and successor blocks.

These methods have little bias because they do not try to fit functions into any preconceived elements. However, they require massive repetition of graph coloring, set covering, or similar techniques to find and evaluate good sets of bound and free variables in every level. Thus, these types of algorithms require speedup. Each of the synthesis or decision algorithms requires solving of an NP problem, such as unate or binate covering. Solving these problems leads to the same form—the product of the sums of literals—as in the well-known satisfiability problem. But these optimization problems are more difficult because they not only check that the formula is satisfied, but must also find the best method to satisfy the formula—by, for instance, assigning a minimum number of non-negated literals to a formula. There are some well-known functions of this type, for example, a product of positive literals called Petrick Function used for binate covering (DNF minimization). Other examples of decision functions are a generalized Petrick function in the form of a product of the sums of products of arbitrary literals and a Helliwell function used in exclusive-OR sum-of-products (ESOP) minimization. Thus, efficient solving and minimizing decision functions in hardware would be useful in a variety of applications.

We use MV logic rather than binary logic in our approach because of its natural representation of multivalued and symbolic concepts.[17,29,32,33,38,39] MV logic algorithms also have high descriptive power and efficiency, as displayed in their use by the newest general-purpose logic synthesis system, MVSIS, from the University of California, Berkeley.[40]

But, ultimately, our approach's goal is to always represent the designed MV network as a binary network implemented in standard binary FPGAs using standard EDA tools. In this way of thinking, binary logic is processed by the system as a special case of MV logic. Moreover, our hardware machine allows for dynamic reprogramming from binary to MV literals with

an arbitrary number of values. This strategy preserves the power of binary logic in applications that need it, and, if necessary, can freely mix binary and MV logic in the same expressions.

Stages

Our approach to learning hardware consists of four stages.

Network construction. A set of positive and negative examples specifies a problem—for example, a certain robot behavior. We consider this set of examples (or cares) as specification of an incompletely specified mapping (in general, a function or a relation). We then select a type of network structure to synthesize, and the system selects an adequate synthesis algorithm. This structure can be a tree, DNF, multilevel decomposed network, or other structure. All available examples feed into the hardware system as MV-input product terms with true or false output values. A virtual-hardware processor now synthesizes a combinational network. This process leads to a generalization of the presented set of examples. Thus, in the synthesis process, from the point of view of an input-output mapping, the initially specified mapping's don't cares become cares in the synthesized network. The don't cares (also called *don't knows*, which are input combinations not given as examples) are replaced with values 0 or 1. Extrapolating don't cares to cares corresponds to learning, generalization, or prediction, depending on the example data. This generalization process has a certain bias, because of the network structure selection and gate type selection in this structure. The bias is small for functional (Ashenhurst) decomposition, because this method does not make any assumption on the structure or gate types. In standard logic synthesis for VLSI circuits, the percent of don't cares is low, however, the practical data in logic synthesis for learning often have more than 99 percent of the don't cares. So algorithms that are efficient for VLSI EDA tools are not always the best ones for learning.

Instead, we can apply several logic synthesis methods—such as the DNF,[31] ESOP,[33] or hierarchical and iterative Ashenhurst/Curtis decomposition,[24,32,41,42] implemented in a hardware/software reconfigurable system.

Network hardware compilation. This stage maps the quasioptimally constructed network from the first stage to a placed and routed network of standard FPGA CLBs. We designed this network using standard partitioning, placement, and routing algorithms realized in EDA software tools from Xilinx or other companies. These tools run in software, and therefore the problem instances given to them should always be regular and small. Otherwise, the software's slow speed would make physical design a bottleneck of the entire hierarchical learning/design approach.

FPGA reconfiguration. By this stage, the knowledge gained by the system is in a set of networks—*knowledge circuits*—corresponding to various behaviors. This knowledge representation is similar to that in subsumption architectures,[43] but in our case, the circuit's knowledge is acquired automatically. This knowledge is stored in binary memory patterns representing virtual circuits. Under the software program's supervision, the hardware can switch between several synthesized circuits (for example, between different robot behaviors).

Execution and network resynthesis. As the system solves new problems—that is, evaluates its knowledge circuits on new minterms—it accumulates the new data sets and training decisions, and the systematic procedures repeatedly redesign the network. In this scenario, an old network can serve as a redesign plan for a new network. For instance, while a procedure synthesizes a multilevel network of CLBs using Ashenhurst decomposition, it can store the sets of bound and free variables for every decomposition level for reuse. In another approach for problems with many new examples, the procedures redesign the network from scratch. This can be done from time to time, to avoid the bias of learning history.

It follows then, that we can replace the evolvable-hardware model of creating high-level behaviors by evolving low-level hardware resources with the learning model at a high level of logic and then compiling the learned logic network to low-level hardware, using standard FPGA-targeted tools. The same physical FPGA resources can be multiplexed to implement the virtual human-designed learning hardware and the automatically designed learned-data hardware (knowledge circuits).

Although we can design the learning hardware once and not change it, the learning-hardware approach can modify the data hardware indefinitely as the environment changes.

Logic operation speedup

Learning hardware development seeks to express a class of important combinatorial problems by operations of the MV cube calculus (MVCC). We then want to efficiently implement the MVCC operations in a specialized hardware processor and compute solutions for these problems with parallel structures of the specialized processors.

The MV-input, binary-output, cube calculus[12,28,38,44-46] could act as a general data representation and calculus for low-level data in several areas, including propositional logic, logic synthesis, logic programming, logic simulation, machine learning, decision making, image processing, databases,[47] set logic, rough sets, partition theory and applications,[48] and a few other areas of problem solving and artificial intelligence[19,29,44,49-51] that represent discrete functions and relations. These functions and relations can be binary, MV, or symbolic. It's possible to develop a general-purpose symbolic processor for efficient computation in all these fields with a large subset of MV-logic operations implemented in hardware. But such a processor would be very complex and expensive, because the number of possible MV-logic operations grows as the number of logic values increases. Moreover, executing each algorithm would use only a small specific subset of all the implemented operations.

Fortunately, modern FPGA technology allows for

- fast and relatively inexpensive realization of specialized programmable processors for arbitrary operations, and
- reuse of the same FPGA hardware to implement various specialized application-specific processors (reprogrammability).

Moreover, FPGA implementations are especially efficient for regular and scalable architectures.

Therefore, we designed and implemented a broad-spectrum reconfigurable hardware accelerator for MV-logic operations. This MVCC processor is also called a cube calculus machine (CCM). CCMs, which we discuss in Part 2 of this article, run very efficiently on FPGAs, because their architecture is highly regular and scalable. A CCM is suitable as an accelerator for a standard PC or a robot microcontroller. It's a hardware accelerator that users can reconfigure to execute a specific algorithm that is required at a certain time. When executing a certain algorithm, the CCM hardware is programmed for only the MVCC operations actually used in the algorithm. Reprogrammability is crucial for effective and efficient implementation of the MVCC to address MVCC logic operation growth but the CCM uses only a small specific subset of all operations for a particular algorithm. Moreover, different MVCC operations have the same general computation pattern. Identifying this pattern let us develop a general processor architecture for all MVCC operations, each involving a different combination of elementary logic operations. Only these elementary logic operations need reprogramming each time the operation type or the number of logic values changes.

The CCM's heart is a bit-sliced data path representing a one-dimensional iterative network of electrically programmable and externally controlled cellular automata. The data path is thus not combinational, but sequential. The internal states of the iterative data path correspond to stages of calculating the resultant cubes. Each sequential MVCC operation generates several words as its result. Figure 1a (next page) shows this most basic type of CCM.

We can realize further speedup by using hardware implementations of typical and repeatedly performed operations, and by implementing appropriate parallelism. Algorithms typically perform operations on a large amount of data. In particular, an algorithm's lowest-level loop—the variable loop—might involve a lot of binary or MV variables.

From the viewpoint of computation speed, it's best to implement computations for all loop variables in parallel by making the word of a single CCM processor at least as long as required by all problem variables. However, each particular algorithm might require different word lengths, which might also vary for a particular execution on particular data. Therefore, the CCM architecture is fully scalable.[49]

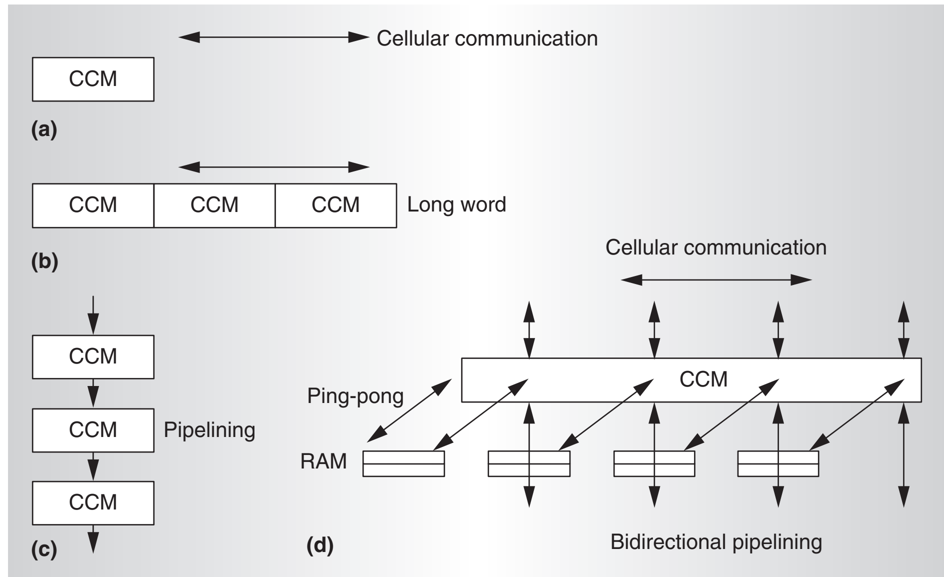

Figure 1. Sample data flows in various structures built from CCM building blocks: a single processor with cellular automata architecture to process standard cubes (a), a long word of several horizontally connected processors to process long cubes (b), pipelining in vertically connected processors to perform 2D array computations (c), and 3D processing using memories (d).

Moreover, to overcome the limitations on physical dimensions for FPGAs and boards with FPGAs, the CCM's design—in particular the iterative cell (IT) and control unit design[49]—lets the horizontal connection of any required number of CCM processors implement long words that exceed the word length of a single CCM, as Figure 1b shows. For each pair of CCM processors in the horizontal chain, the control unit enables direct connection of the rightmost IT of the left processor with the leftmost IT of the right-hand processor of the pair, without using the bus interface unit (BIU). These two CCM architecture features implement microparallelism at the subinstruction level.

We can further enhance the computation speed by using a CCM-processor vertical linear array (pipeline) to implement the algorithm's second-lowest-level loop—usually the cube loop or instruction loop. The CCM's BIU design enables communication vertically among CCM processors and among CCM processors and the host processor.[49] The BIU also enables bidirectional pipelining of CCMs and ping-pong communications between any CCM processor and RAM; these processes aid some computations involving cubes. For example, the architecture involving both the two-directional pipelining and ping-ponging can compute the morphological Hough transform, which some robots use to find shapes and their locations. Figure 1c shows a pipelined, vertical array of CCMs.

Besides these massively parallel CCM structures, other possible structures include the tree of pipelined CCMs, which is useful for many computations—for example, computing the generalized Petrick function for solving the unate covering problem.[28] Such flexibility lets CCM virtual processors realize 3D massively parallel processing architectures with 3D data movements. Figure 1d shows one such 3D configuration.

CCMs present opportunities for implementing massively parallel structures with reprogrammable FPGAs, which leads to a high degree of flexibility. The same reprogrammable FPGA hardware can implement various massively parallel structures made up of differently programmed CCM processors, and also various other coprocessors or accelerators. Moreover, the number of possible-to-program CCM operations is high. These operations include all MVCC operations, MV multioutput relations

operations, simplified calculus operations in decomposition, another simplified calculus used in rough-set theory and rough partitions, symbolic predicates, and so on. Moreover, in various CCM processors or a single CCM's various iterative logic units, users can concurrently program different MVCC operations for MV variables with different numbers of values. Part 2 of this article, starting on page 52, gives an introduction to cube calculus and describes the simplified CCM architecture. MICRO

References

1. F. Kocan and D.G. Saab, "Concurrent D-Algorithm on Reconfigurable Hardware," *Proc. IEEE/ACM Int'l Conf. Computer-Aided Design* (ICCAD), ACM Press, New York, 1999, pp. 152-156.
2. M. Abramovici and J.T. de Sousa, "A SAT Solver Using Reconfigurable Hardware and Virtual Logic," *SAT2000, Highlights of Satisfiability Research in the Year 2000*, *Frontiers in Artificial Intelligence and Applications*, IOS Press, Amsterdam, pp. 377-402.
3. J. Cong and J. Peck, *On Acceleration of Logic Synthesis Algorithms Using FPGA-Based Reconfigurable Coprocessors*, tech. report TR 970010, Computer Science Dept., Univ. of California, Los Angeles, 1997.
4. T. Hanyu, T. Kodama, and T. Higuchi, "200-Vertex On-Chip Clique-Finding VLSI Processor for Real-Time 3D Object Recognition," *IEEE Int'l Conf. Industrial, Electronics, Control, Instrumentation and Automation*, vol. 3, IEEE Press, Piscataway, N.J., 1992, pp. 1379-1384.
5. H. DeGaris, "Evolvable Hardware: Genetic Programming of a Darwin Machine," *Artificial Nets and Genetic Algorithms*, R.F. Albrecht, C.R. Reeves, and N.C. Steele, eds., Springer-Verlag, Heidelberg, Germany, 1993, pp. 441-449.
6. H. de Garis, *CAM-Brain Machine (CBM) FPGA Design*; http://www.cs.usu.edu/~degaris/cbm/index.html.
7. H. de Garis et al., "Building an Artificial Brain Using an FPGA Based CAM-Brain Machine," *Applied Mathematics and Computation J.*, Special Issue on Artificial Life and Robotics, Artificial Brain, Brain Computing and Brainware, vol. 111, 2000, North Holland, Amsterdam, pp. 163-192.
8. T. Higuchi, M. Iwata, and W. Liu, eds., *Proc. 1st Int'l Conf. Evolvable Systems*, *Lecture Notes in Computer Science*, no. 1259, Springer-Verlag, Heidelberg, Germany, 1997.
9. A. Nicholson, "Evolution and Learning for Digital Circuit Design," *Proc. Genetic and Evolutionary Computation Conf.* (GECCO), Morgan Kaufmann, San Francisco, 2000, pp. 519-524.
10. M.A. Perkowski, A.N. Chebotarev, and A.A. Mishchenko, "Evolvable Hardware or Learning Hardware? Induction of State Machines from Temporal Logic Constraints,'' Proc. 1st NASA/DOD Workshop Evolvable Hardware, A. Stoica, D. Keymeulen, and J. Lohn, eds. IEEE CS Press, Los Alamitos, Calif., 1999, pp. 129-138.
11. M. Perkowski, "Systolic Architecture for the Logic Design Machine," *Proc. IEEE/ACM Int'l Conf. Computer Aided Design* (ICCAD), ACM Press, New York, 1985, pp. 133-135.
12. M.A. Perkowski, "A Universal Logic Machine," *Proc. 22nd IEEE Int'l Symp. Multiple Valued Logic* (ISMVL), IEEE CS Press, Los Alamitos, Calif., 1992, pp. 262-271.
13. M. Perkowski et al., "Software-Hardware Codesign Approach to Generalized Zakrevskij Staircase Method for Exact Solutions of Arbitrary Canonical and Non-Canonical Expressions in Galois Logic," *Booklet of 6th Int'l Workshop Post-Binary ULSI Systems*, T. Hanyu, ed., St. Francis Xavier Univ., Antigonish, N.S., Canada, 1997, pp. 41-44.
14. M.A. Perkowski, L. Jozwiak, and D. Foote, "Architecture of a Programmable FPGA Coprocessor for Constructive Induction Approach to Machine Learning and other Discrete Optimization Problems," *Reconfigurable Architectures. High Performance by Configware*, R.W. Hartenstein and V.K. Prasanna, eds., IT Press Verlag, Bruchsal, Germany, 1997, pp. 33-40.
15. R.L. Ashenhurst, "The Decomposition of Switching Functions," vol. 29, *Annals Harvard Computer Laboratory*, Harvard Univ., Cambridge, Mass., 1959, pp. 74-116.
16. H.A. Curtis, *A New Approach to the Design of Switching Circuits*, Van Nostrand, Princeton, N.J., 1962.
17. R.S. Michalski and J.B. Larson, "Inductive Inference of VL Decision Rules," *ACM SIGART Newsletter*, no. 63, June 1977, pp. 38-44.

18. R.S. Michalski, I. Bratko, and M. Kubat, *Machine Learning and Data Mining: Methods and Applications*, John Wiley and Sons, New York, 1998.
19. Z. Pawlak, *Rough Sets: Theoretical Aspects of Reasoning about Data*, Kluwer Academic, Boston, 1991.
20. Y. Abu-Mostafa, ed., *Complexity in Information Theory*, Springer-Verlag, New York, 1988, p. 184.
21. C. Files and M. Perkowski, "An Error Reducing Approach to Machine Learning Using Multi-Valued Functional Decomposition," *Proc. Int'l Symp. Multiple-Valued Logic* (ISMVL), IEEE CS Press, Los Alamitos, Calif., 1998, pp. 167-172.
22. C. Files and M. Perkowski, "Multi-Valued Functional Decomposition as a Machine Learning Method," *Proc. Int'l Symp. Multiple-Valued Logic*, IEEE CS Press, Los Alamitos, Calif., 1998, pp. 173-178.
23. C. Files and M. Perkowski, "New Multivalued Functional Decomposition Algorithms Based on MDDs," *IEEE Trans. CAD*, vol. 19, no. 9, Sept. 2000, pp. 1081-1086.
24. R. Malvi, M. Perkowski, and L. Jozwiak, "Exact Graph Coloring for Functional Decomposition: Do We Need it?," *Proc. 3rd Int'l Workshop Boolean Problems*, B. Steinbach, ed., Tech. Univ. Freiberg, Germany, 1998, pp. 1-10.
25. T.D. Ross et al., *Pattern Theory: An Engineering Paradigm for Algorithm Design*, tech. report WL-TR-91-1060, Wright Laboratories, USAF, Wright Patterson Air Force Base, Ohio, 1991.
26. T. Sasao, ed., *Representation of Boolean Functions*, Kluwer Academic, Boston, 1996.
27. G. deMicheli, *Synthesis and Optimization of Digital Circuits*, McGraw-Hill, New York, 1994.
28. D.L. Dietmeyer, *Logic Design of Digital Systems*, Allyn and Bacon, Boston, 1971.
29. D. Rine, *Computer Science and Multiple Valued Logic: Theory and Applications*, North-Holland, Amsterdam, 1984.
30. D. Michie, "Machine Learning in the Next Five Years," *Proc. 3rd European Working Session on Learning* (EWSL), Pitman, London, 1988, pp. 107-122.
31. L. Nguyen, M. Perkowski, and N. Goldstein, "PALMINI—Fast Boolean Minimizer for Personal Computers," *Proc. IEEE/ACM 24th Design Automation Conf.*, ACM Press, New York, 1987, pp. 615-621.
32. M. Perkowski et al., "Decomposition of Multiple-Valued Relations," *Proc. IEEE Int'l Symp. Multiple Valued Logic* (ISMVL), IEEE CS Press, Los Alamitos, Calif., 1997, pp. 13-18.
33. M.A. Perkowski et al., "Application of ESOP Minimization in Machine Learning and Knowledge Discovery," *Proc. 2nd Workshop Applications of Reed-Muller Expansion in Circuit Design*, Fujiki Printing Co, Iizuka, Japan, 1995, pp. 102-109.
34. K. Dill and M. Perkowski, "Minimization of Generalized Reed-Muller Forms with Genetic Operators," *Proc. Genetic Programming*, Morgan Kaufmann, San Francisco, 1997, p. 362.
35. K. Dill and M. Perkowski, "Baldwinian Learning Utilizing Genetic and Heuristic Algorithms for Logic Synthesis and Minimization of Incompletely Specified Data with Generalized Reed-Muller (AND-EXOR) Forms," *J. Systems Architecture*, vol. 47, no. 6, June 2001, pp. 477-489.
36. K. Dill, J. Herzog, and M. Perkowski, "Genetic Programming and its Application to the Synthesis of Digital Logic," *Proc. IEEE Pacific Rim Conf. Comm. and Signal Processing* (PACRIM), IEEE Press, Piscataway, N.J., 1997, pp. 823-826.
37. L. Jozwiak and A. Postula, "Genetic Engineering versus Natural Evolution: Genetic Algorithms with Deterministic Operators," *Proc. Int'l Conf. Artificial Intelligence* (IC-AI), CSREA Press, 1999, pp. 58-64.
38. S.Y.H. Su and P.T. Cheung, "Computer Minimization of Multi-Valued Switching Functions," *IEEE Trans. Computers*, vol. C-21, no. 9, 1972, p. 995.
39. B. Zupan and M. Bohanec, *Learning Concept Hierarchies from Examples by Function Decomposition*, tech. report, Dept. Intelligent Systems, Josef Stefan Inst., Ljubljana, Slovenia, 1996.
40. M. Gao et al., "Optimization of Multi-Valued Multi-Level Networks," *Proc. Int'l Symp. Multiple-Valued Logic* (ISMVL), IEEE CS Press, Los Alamitos, Calif., 2002, pp. 168-177.
41. M. Perkowski et al., "Graph Coloring Algorithms for Fast Evaluation of Curtis Decompositions," *Proc. 36th ACM/IEEE Design Automation Conf.* (DAC), ACM Press, New York, 1999, pp. 225-230.

42. PSU POLO Directory with DM/ML Benchmarks; http://www.ece.pdx.edu/~polo/.
43. R.A. Brooks, "A Robust Layered Control System for a Mobile Robot," *IEEE J. Robotics and Automation*, vol. 2, no. 1, Mar. 1986, pp. 14-23.
44. L. Kida and M.A. Perkowski, "The Cube Calculus Machine: A Ring of Asynchronous Automata to Process Multiple-Valued Boolean Functions," *Proc. IEEE Int'l Symp. Circuits and Systems* (ISCAS), IEEE Press, Piscataway, N.J., 1992, pp. 807-810.
45. T. Sasao, "HART: A Hardware for Logic Minimization and Verification," *Proc. IEEE Int'l Conf. Computer Design* (ICCD), IEEE CS Press, Los Alamitos, Calif., 1985, pp. 713-718.
46. N. Song and M. Perkowski, "Minimization of Exclusive Sum-of-Products Expressions for Multiple-Valued Input, Incompletely Specified Functions," *IEEE Trans. CAD*, vol. 15, no 4, 1996, pp. 385-395.
47. E.F. Codd, "A Relational Model of Data for Large Shared Data Banks," *Comm. ACM*, vol. 13, no. 6, June 1970, pp. 377-387.
48. S. Grygiel and M. Perkowski, "Labeled Rough Partitions—A New General Purpose Representation for Multiple-Valued Functions and Relations,'' *J. Systems Architecture*, vol. 47, no. 1, Jan. 2001, pp. 29-59.
49. Q. Chen, *Realization of a Universal Cube Calculus Machine in DECPeRLe-1 FPGA Emulator*, master's thesis, Dept. of Electrical and Computer Engineering, Portland State Univ., Ore., 1998.
50. S. Devadas, "Optimal Layout via Boolean Satisfiability," *Proc. Int'l Conf. Computer Aided Design* (ICCAD), IEEE CS Press, Los Alamitos, Calif., 1989, pp. 294-297.
51. D. Foote, *The Design, Realization and Testing of the ILU of the CCM2 Using FPGA Technology*, master's thesis, Dept. of Electrical Eng., Portland State Univ., Ore., 1994.

Biographies for **Marek Perkowski**, **David Foote**, **Qihong Chen**, **Anas Al-Rabadi**, and **Lech Jozwiak** are on p. 61 of this issue.

Direct questions and comments about this article to Marek Perkowski, Portland State Univ., Dept. of Electrical and Computer Engineering, Portland, OR 97207; mperkows@ece.pdx.edu.

Let your e-mail address show your professional commitment.

An IEEE Computer Society e-mail alias forwards e-mail to you, even if you change companies or ISPs.

you@computer.org

The e-mail address of computing professionals