LEARNING HARDWARE USING MULTIPLE-VALUED LOGIC

Part 2: Cube Calculus and Architecture

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 APPROACH IN DIGITAL-CIRCUIT-DESIGN AUTOMATION.

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

Lech Jozwiak
Eindhoven University of Technology

This article proposes using symbolic learning methods based on multiple-valued (MV) logic and implemented in reconfigurable hardware. In the part one, we discussed why symbolic learning is useful in some applications, such as robotics. We presented an architecture for a massively parallel reconfigurable processor that enables speeding up logic operations performed in learning hardware.

Rather than learning using evolutionary and neural network methods in hardware, our approach uses combinatorial synthesis methods developed in the framework of the logic synthesis approach in digital-circuit-design automation. In contrast to previous approaches to evolvable hardware that so far have dominated the learning in reconfigurable systems, here the learning takes place on the level of constraint acquisition and quasioptimal logic synthesis rather than on the lower level of programming binary switches based on (close to random) decisions of evolutionary programming methods. Our learning strategy is based on the principle of Occam's Razor, which facilitates generalization, discovery, and strong learning methods. We realize directly in reconfigurable hardware such MV cube-algebra operators as intersection, supercube, sharp, and crosslink, and such algorithms as disjunctive normal form (DNF) minimization, Ashenhurst/Curtis decomposition, satisfiability, and decision tree generation. Part two of our article presents cube calculus in more detail as well as various aspects of realizing cube calculus operations in hardware. We also evaluate two variants of our experimental designs.

Cube calculus

Let's consider discrete variables (attributes) $X_1, X_2, \ldots, X_n$, such that each variable X_i can take values from a certain finite discrete set V_i (V_i can be any finite set of symbols). A literal X_i^{Si} of variable X_i represents a characteristic function of subset S_i of V_i, that is, the literal's value is 1 for symbols from this subset. For example,

- for binary logic, $X^1 = X$, and $X^0 = X'$ are two literals; and
- for four-valued logic $V_i = \{0, 1, 2, 3\}$: $Y^{\{0, 2\}}$ equals 1 if $Y \in \{0, 2\}$ or 0 if $Y \in \{1, 3\}$ is a literal.

A cube on $X_1, X_2, \ldots, X_n$ is an ordered set of literals on $X_1, X_2, \ldots, X_n$, that is, $X_1^{S1} X_2^{S2} \ldots X_n^{Sn}$.

A cube represents a subspace in the n-dimensional discrete MV space, as Figure 1 shows. Usually, traditional switching algebra

0272-1732/02/$17.00 © 2002 IEEE

interprets a cube as a product of literals. Here we also interpret it as a sum of literals or an XOR of literals. In general, a cube can represent any ordered set of subsets of certain discrete sets V_i, $i = 1, \ldots, n$ (that is, a Cartesian product of the subsets).

Cube calculus is a system of

- a set of all cubes on a certain ordered set of discrete variables $X_1, X_2, \ldots, X_n$ that also contains an empty cube and a full cube (full discrete space defined by $X_1, X_2, \ldots, X_n$); and
- a set of operations on sets of cubes.

There are several types of cube operations:

- *Cube operators* result is a list of 0 to n cubes.
- *Cube predicates* result is the logic values 0 or 1.
- *Counting operations* result is a number (for instance, the Hamming distance of two cubes).

Positional notation represents literals in the cube calculus machine (CCM). Positional notation uses a separate bit to represent each possible value of each literal. If the literal is true for a specific value, the corresponding bit is set to 1. For example, assuming that each of the variables X_1, X_2, and X_3 is a three-valued variable, and the values are {0, 1, 2}, the cube $X_1^{\{0,2\}}X_2^{\{1,2\}}X_3^{\{2\}}$ will be denoted by [101 011 001]. The base K of a logic machine is the number of bits required to represent a simple symbol in this machine.

For example, $K = 2$ realizes all logic operations in MV logic with no more than $2^2 = 4$ values. Every operator can be represented by its map. The four simple symbols are 0 for a negated variable, 1 for a positive variable, X for don't care, and ε for contradiction. We encode these symbols in positional notation as follows: 0 = 10, 1 = 01, X = 11, and ε = 00. With this encoding, cube bcd′ (X110) on the ordered set of discrete variables (*a*, *b*, *c*, and *d*) is represented by [11 01 01 10].

When we realize MV logic using binary signals, K binary signals represent each simple symbol from the set of 2^K symbols, as Figure 2a (next page) shows. A simple symbol expressed in base K is a fundamental idea of our machine.

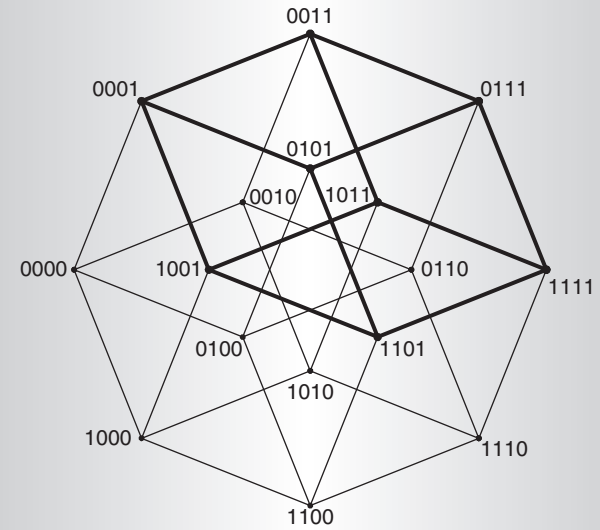

Figure 1. A 3D cube (XXX1) in a 4D space (XXXX). Symbol X represents a value 0 or 1. Thus cube XXX1 denotes a cube variable X_4 and XXXX denotes value 0 or 1 for each of four variables, thus the entire 4D space.

A symbol of base $K = 1$ is sufficient to realize the binary logic, set theory, and binary arithmetic. A K of 2 is necessary for binary cube calculus and some MV systems (shown in Figure 2b). Figure 2c shows programmability matrix for three binary arguments and $K = 2$. For more operations on MV systems, K must be greater than 2. A W-input, base K universal cell is a logic block with W inputs and one output, each input and output being a base-K signal (Figure 2d). A big enough K can realize any operator, and this is why we call this the universal cell realizing operations of universal logic.

The iterative cell (IT) processes each simple symbol in a CCM, since the IT is an elementary processor. A K-base symbol requires a K-base IT cell, shown in Figure 3 for $K = 2$. In a CCM of base K, each variable can have an arbitrary, but divisible by K, number of values. R cooperating ITs process a complex variable with $R \times K$ values (a complex symbol). Thus, CCMs introduce simple and complex symbols as an intermediate level between bits and variables, enabling a flexible number of values in literals. They also permit the use of, for example, a 2-bit IT cell to represent part of an MV literal for an arbitrary number of values.

The main concepts behind hardware implementation of the cube calculus data are

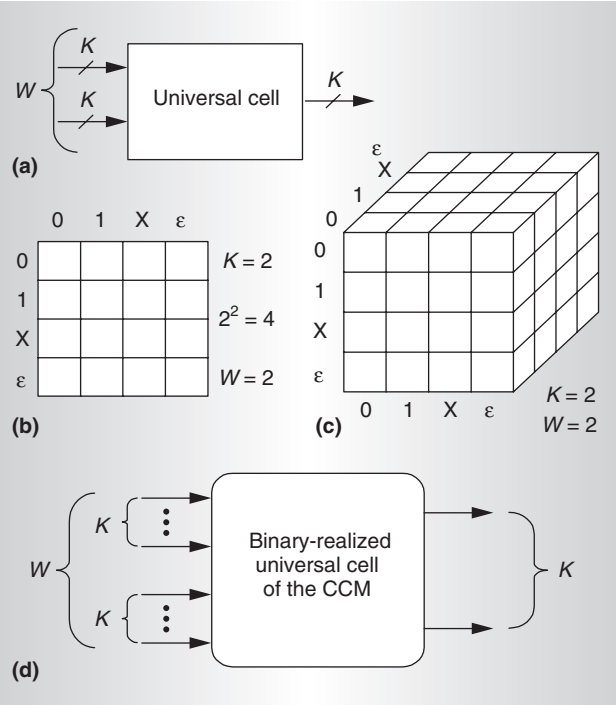

Figure 2. *W*-input, *K*-base universal cells: universal cell with two 2^K-valued inputs that realizes all matrices of base 2^K (a). The matrix represents all operations for $K = 2$ and two arguments; it uses the four symbols of binary cube calculus. Filling the cells in all possible ways with symbols 0, 1, *X*, and ε creates new operations (b). We can build a similar programmability matrix for three binary arguments (c) and a universal cell for all *K*-output Boolean functions of $K \times W$ input variable (d).

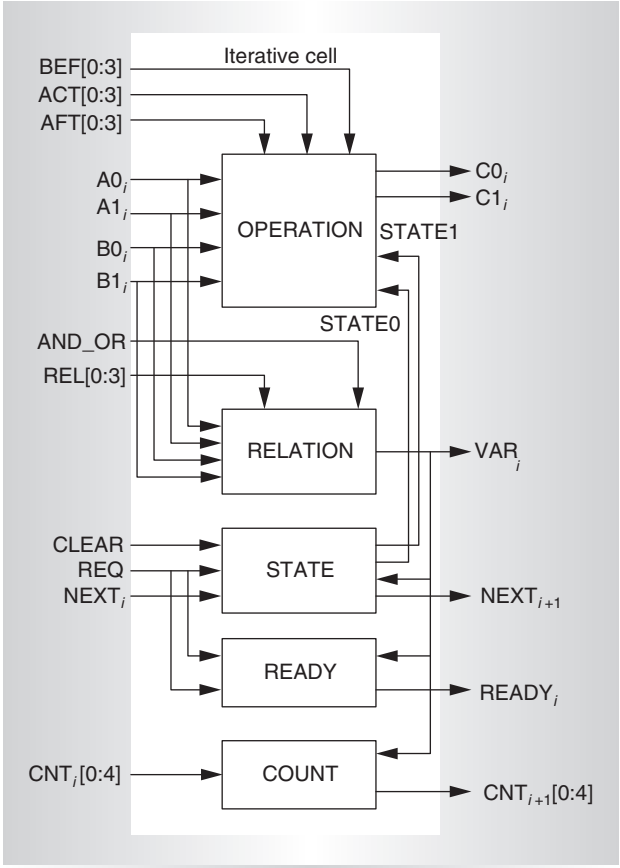

Figure 3. Iterative cell that operates on 2 bits. It is a single finite state machine in a cellular automaton. We explain the meaning of the most important signals—BEF, ACT, or AFT—in the text. These signals program each operation executed in phases of the cube calculation according to the internal state of machine STATE. AND_OR selects the type of relations checked on data, COUNT executes counting operations, STATE records the internal states of the data path that correspond to operations BEF, ACT, and AFT.

- a cube,
- a list (array) of cubes (clist), and
- a list of lists of cubes (cclist).

Hence, MV cube calculus (MVCC) is a set of cubes of a certain discrete space and a set of operations on the cubes, clists, and cclists.

Computation patterns

There are many (two-argument) operators on cubes, but fortunately they expose certain common computation patterns, which we can subdivide into three classes:

- simple combinational operations (such as intersection),
- complex (conditional) combinational operations (such as prime), and
- sequential operations (such as nondisjoint sharp).

Operations of each class have the same computation pattern, and the pattern of a simpler class is a special case of a more complex class; Table 1 shows these patterns. The common computation pattern enables the design of certain dedicated-hardware architectures for cube operations. We implement each particular operation using a particular instantiation of the general architecture, realized by appropriately programming the field-

Table 1. Computational patterns for REL, BEF, ACT, and AFT.

	Relation (REL)		Output operation		
Function	**REL**	**and/or**	**Before (BEF)**	**Active (ACT)**	**After (AFT)**
Intersection	1	and	Ai ∩ Bi	—	—
Supercube	1	and	Ai ∪ Bi	—	—
Prime	Ai ∩ Bi ≠ ∅	and	Ai	Ai ∪ Bi	
Crosslink	Ai ∩ Bi = ∅	and	Ai	Ai ∪ Bi	Bi
Sharp	(Bi ⊇ Ai)	or	Ai	Bi ∩ Ai	Ai
Disjoint sharp	(Bi ⊇ Ai)	or	Ai	Bi ∩ Ai	Ai ∩ Bi
Symmetric consensus	1	and	Ai ∩ Bi	Ai ∪ Bi	Ai ∩ Bi
Asymmetric consensus	(Bi ⊇ Ai)	or	Ai ∩ Bi	Ai ∪ Bi	Ai ∩ Bi

programmable gate array (FPGA) structures.

Simple combinational operations

- produce a single cube as a result and
- are position-by-position operations, that is, they use the same operation for each position.

Complex (conditional) combinational operations

- produce a single cube as a result,
- are position-by-position (bit-by-bit) operations,
- calculate the literals of the resulting cube from the corresponding literals of the argument (operand) cubes by conditional operations on the literals of the argument cubes.

Sequential operations can

- produce more than one resulting cube, that is, one cube per active literal (active position);
- have from 0 to n potentially active positions i that satisfy a certain relation REL(A_i, B_i);
- execute operations on the active position with a certain simple combinational operation;
- execute operations before and after the active position (in general) with certain simple combinational operations, although these often consist of just copying A_i or B_i;
- simultaneously compute potentially active positions; and
- sequentially create the resulting cubes starting from the leftmost potentially active position.

Simple combinational operation example

The following is an intersection (product) operation on cubes A and B:

$$A \cap B = \begin{cases} \varnothing \text{ if } A_i \cap B_i = \varnothing \text{ for some } i \\ A_i \cap B_i, A_2 \cap B_2, \ldots, A_n \cap B_n \text{ otherwise} \end{cases}$$

Complex combinational operation example

The following is a prime operation:

A prime B =

$$X_1^{A_1} X_2^{A_2} \ldots X_{i-1}^{A_{i-1}} X_i^{A_i \cup B_i} X_{i+1}^{A_{i+1}} \ldots X_n^{A_n}$$

where $A_i \cup B_i$ are computed only for those variables X_i for which $A_i \cap B_i \neq \varnothing$ is satisfied (such positions are called active positions).

Sequential operation example

The following is a nondisjoint sharp

$$A \# B \begin{cases} A \text{ when } A \cap B = \varnothing \\ \varnothing \text{ when } A \subseteq B \\ A \#_{basic} B \text{ otherwise} \end{cases}$$

$A\#_{basic}B = \{X_1^{A_1} X_2^{A_2} \ldots X_{i-1}^{A_{i-1}} X_i^{\neg A_i \cap B_i} X_{i+1}^{A_{i+1}} \ldots X_n^{A_n} |$ for such $i = 1, \ldots, n$ that $\neg (A_i \subseteq B_i)\}$. For instance, XXX1 # 111X = {0XX1, X0X1, XX01}.

It is important to observe that the same general pattern given below can describe every sequential operation:

A operation B = $\{X_1^{\text{aft}(A1,B1)} \ldots X_{i-1}^{\text{aft}(Ai-1,Bi-1)} X_i^{\text{act}(Ai,Bi)} X_1^{\text{bef}(Ai+1,Bi+1)} \ldots X_{i-1}^{\text{bef}(An,Bi-1)}\}$| for such $i = 1, \ldots, n$ that REL(A_i, B_j) is satisfied.

Functions REL, BEF, AFT, and ACT can thus specify every operation. Table 1 shows that for simple and complex combinational operations, only those columns that specify this operation are filled.

Patterns of combinational operations are special cases of this general pattern. Thus, simple combinational operations are a special case of the complex combinational operations, and all combinational operations are a special case of the sequential operations. For different operations, we select different functions for REL, BEF, ACT, and AFT.

Compare the previous operation examples and Table 1. Functions REL, BEF, ACT, and AFT are *K*-wise functions. If, for example, *K* = 2, then each 2 bits of resulting cube C that represent a simple symbol depend only on the corresponding 2 bits of argument cubes A and B, that is, C_iC_{i+1} depend only on A_iA_{i+1} and B_iB_{i+1}. A complex symbol that represents a value of variable *C* that has length $R \times K$(IT) is a composition of *R* simple symbols and represents the results computed in all ITs representing this variable. The ITs form the *R K*(IT)-sliced CCM chips. Example applications of these operations include intersection, the most common operation in standard binary logic, rough set, and decomposition calculations; supercube, used in DNF minimization and ESOP synthesis; prime, used in XOR-based synthesis; nondisjoint sharp, used in tautology, satisfiability, covering, and DNF algorithms; disjoint sharp, used in representation transformations; and consensus, used in automatic theorem proving (Prolog's subset implementation) and DNF minimization.

CCM architecture

Software implementation of each cube operation uses a single loop that runs through all cube variables. The following two crucial ideas form the basis for the CCM:

- Execution of the lowest-level loop of the cube operation algorithms—the variable loop—occurs in hardware using a linear, iterative array of cellular automata (finite state machines). Information flows between the FSMs from left to right and from right to left. Every FSM can be in internal states BEF, ACT, and AFT. This state selects the corresponding BEF, ACT, or AFT function programmed to an iterative logic unit (ILU).
- Reconfiguration of logic functions of the cellular automata is achieved by their implementation with FPGAs.

The CCM system architecture involves

- a host processor, since it is a traditional general-purpose computer; and
- the massively parallel array of the CCM processors, which forms a coprocessor (application-specific reconfigurable hardware accelerator) of the host processor.

A single CCM processor[1] consists of

- an ILU, which is a horizontal linear array of *R K*(IT)-sliced CCM processors, each composed of *R* ITs, as Figure 4 shows;
- a control unit that controls ILU operation and executes cube calculus operations;
- a register file to store auxiliary and control registers that aid in operations; and
- a bus-interface unit (BIU) to control internal and external flow of cube array data among processors, host, and memories.

The lowest-level loop—usually the variable loop—is implemented inside the CCM processors by horizontal communication between the ITs, or, with a few CCM processors, which are connected horizontally. The CCM processors' vertical linear array (pipeline) implements the second lowest level loop—usually the cube loop. This enables 2D-data movement: horizontal (inside or among CCM processors) and vertical (among CCM processors). RAM memories connected to ITs realize the third dimension of data movements. Figure 1 of Part 1 (see page 48) gives some possible data flows in structures built from CCM building blocks.

The implementation of predicates, counting operations, and combinational operators in CCMs is similar to their implementation in a traditional arithmetic processor. Sequential-operator implementation requires more discussion: First, literals at position *i* must satisfy

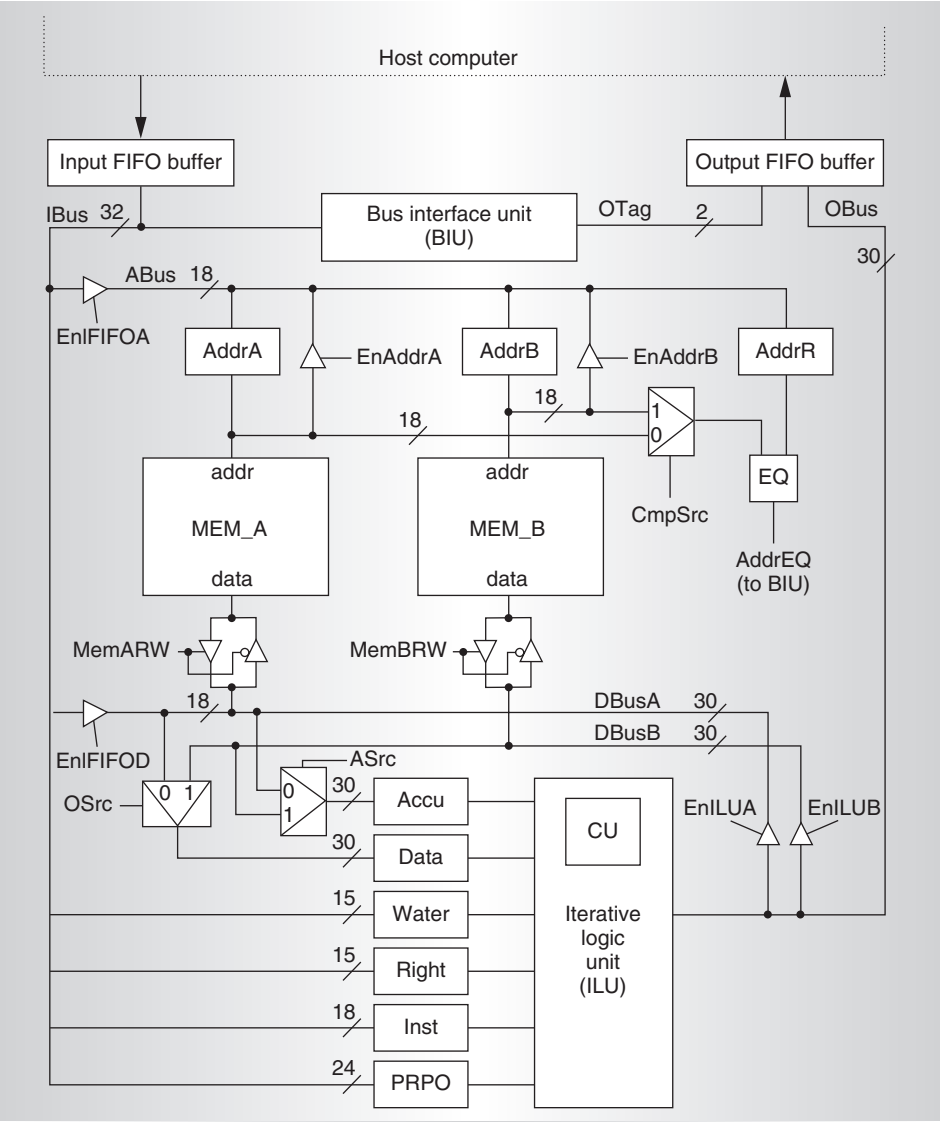

Figure 4. Simplified CCM processor. The figure shows two memory banks, various buses, a bus control unit, a control unit, and registers. The Right and Water registers determine ranges of multivalued variables composed of IT units and help to propagate values through ITs not used while cubes are shorter.

relation REL(A_i, B_i) to activate this position and create a resultant cube. Observe that CCM computes all potentially active positions in parallel during evaluating REL(A_i, B_i) for all i in all ITs. However, the CCM then activates the corresponding literals (positions) sequentially, and sequentially creates exactly one resultant cube for each literal (position) active at a certain time, by executing

- a certain operation ACT (A_i, B_i) at the active literal (for example, $\neg A_i \cap B_i$ for sharp),
- a certain operation BEF (A_i, B_i) at all positions before (or to the right of) the active literal (for example, copying the literals for sharp), and
- a certain operation AFT (A_i, B_i) at all positions after (or to the left of) the active literal (for example, copying the literals for sharp).

Each cube is created in only one clock pulse. The literals are activated starting from the leftmost potentially active position. After per-

forming the previously described computations for a certain active literal, the CCM activates the literal corresponding to the next to the right potentially active position and creates the corresponding resultant cube. When producing a particular resultant cube, all literals on positions to the left of the active literal are of the after type, and all literals on positions to the right from the active literal are of the before type. The iterative network of small, fast FSMs (ITs) executes all of these operations in hardware—using fast communication between the FSMs. This type of controlled cellular automata used as a data path of a general-purpose processor is a new concept in computer architecture.

The number of all possible operations programmed in this way is extremely large. The operations that are possible to program are a multidimensional space created by a Cartesian product of basic programmable features like those shown in Table 1: REL, BEF, ACT, AFT, composition, and pipelining. Every realizable operation is a point in this space. Selection of the operation occurs without reconfiguring the entire data path or control. We consider CCM a prototype symbol-processing computer with a sort of data path microprogramming, in contrast to the control path microprogramming of traditional arithmetic computers.

Advantages

CCMs can greatly speed up many applications. They efficiently implement (multilevel) logic operations unlike a conventional computer's ALU. For instance, to calculate the consensus of two cubes, an ALU must execute a long series of shifts and ANDs. Also, some resultant cubes are empty and require removal, making the generation of resultant cubes irregular and inefficient. The CCM can execute each MVCC operation in a single clock pulse or a few clock pulses; this execution requires only one CCM instruction per operation. The CCM does not generate empty resultant cubes, so resultant-cube generation is regular. The time needed to generate the cubes depends solely on the number of nonempty resultant cubes of the particular operation.

Designers tune traditional, general-purpose information processors (Turing-machine equivalents) for arithmetic computations. In contrast, the CCM, although also a general-purpose processor, is tuned to symbolic computations. The result of a single development process, the CCM can efficiently manage a broad range of applications. It is reprogrammable, which enables implementation of only the operations actually required for a certain algorithm's execution during a certain time slot. It allows a customized instruction set, which programmers can optimize for each application. Moreover, the reconfiguring program on a host computer can reconfigure SRAM-based FPGAs even while host program that uses CCMs is in full operation.

Furthermore, in a conventional computer, a program stored in RAM provides the control. This strategy results in considerable control overhead, because the instructions must be fetched from RAM. If an algorithm contains loops, the processor will read the same instructions many times. This repeated work causes bottlenecks in the memory interface of conventional computer architecture, especially when the memory bus is not as fast as the internal processor bus. In the CCM architecture, the CCM data path itself implements most of the control. Once a complex MVCC instruction is loaded into the CCM, the host computer only needs to write data cubes to the CCM and read the resultant cubes from the CCM. The host processor can process the resultant values from the CCM while loading them; meanwhile, the CCM awaits the next clock pulse to send another cube.

Additionally, in most commonly used computer architectures, parallel processing is very limited, even in modern RISC or Pentium processors. Parallel processing has also proven difficult for compilers. In the CCM architecture, a single CCM instruction can replace parts of an existing program for a traditional computer. Hardware specifically designed for this particular instruction can then execute it, allowing microparallelism in the CCM.

Another limitation of conventional computer architectures is the ALU's bandwidth. The CCM suffers from this problem to a much lesser extent because the FPGA implementation flexibly adopts ALU bandwidth for each application. The only limiting factor is the capacity and speed of the FPGAs in the hardware.

Furthermore, the CCM architecture is regular and scalable, and lets designers build massively parallel computers from many CCM

processors. Other CCMs—and ultimately, the host computer—control these CCM processors. Thus, it's possible to realize true massively parallel processing. Mapping these architectures onto the FPGAs requires considerable time, but once compiled, these new architectures are instantly loadable into the FPGA board.

The question arises as to whether the speedup of a certain application justifies the development or purchase of a costly FPGA board. However, we can spread the cost of FPGA hardware among various applications, not only those involving symbolic computations. Moreover, the essence of the CCM is not the FPGA board, but the architecture programmed into the FPGAs. Because the essence of the CCM is its architecture, which involves reprogrammable, basic logic operations of the ILU for REL, BEF, ACT, and AFT, it's not necessary to implement the CCM on a classical FPGA board. Limiting reprogrammability to only REL, BEF, ACT, and AFT; and/or implementing most of the CCM processor in classical, hardwired hardware can provide faster execution for one of CCM's variant implementations. In this way, designers can implement a CCM processor as a very fast classical VLSI hardware chip with a few small, reprogrammable lookup tables in its ILU.

First prototype evaluation

We have designed, simulated, and implemented a CCM prototype for a word length of 16 binary, eight 4-valued, or four 8-valued variables; or any combination of binary, 4-, or 8-valued variables for a total of 32 bits. Our prototype implementation used two Xilinx FPGA XC-3090-50 PP175C chips with 175 pins and running at 50 MHz. Eight iterative cells of the prototype consumed approximately 48 percent of the available configurable logic blocks (CLBs). We simulated and tested the prototype implementation on many data examples for each operation. We also performed timing analysis: The greatest delay was 145.8 ns. The sharp operation speedup on six variable terms was approximately 25 times that of the software implementation. The speedup of the algorithm for the satisfiability problem was approximately 14 times that of its software implementation. We achieved these speedups on a single CCM processor having an ILU with a short word composed of eight IT cells.

Since speedups grow with the number of IT cells in a single CCM processor and with the number of CCM processors in various massively parallel architectures, computation speed enhancement in the full-scale massively parallel implementation should be much higher. To develop an idea of the possible speed enhancement, we performed several simulation experiments with a tree of pipelined CCM processors used for computation of the generalized Petrick function (this function, being a product of sums of literals, solves the unate covering problem). Among others, we considered a small tree with three levels and seven CCM processors. We assumed a host processor clock rate of 100 MHz. Because the host processor must fetch all four leaf nodes of the CCM processor tree in every execution cycle, it limits the clock rate of the tree to a slow 2.5 MHz. With these assumptions, the considered (small) parallel processing structure solves a generalized Petrick function with 1,000 sums within 0.7 ms. To solve this problem with the same algorithm implemented in C, a traditional host computer (PC) operating at 100 MHz requires 7.08 seconds. Thus, application of an appropriate parallel structure of the CCM processors, even with a small number of processors, resulted in an approximately 10,000 times speedup.

Second prototype evaluation

For the second evaluation we used the DEC-PERLE-1 board.[2] This has a central computational matrix composed of 16 Xilinx XC3090 logic cell arrays, surrounded by four 1-Mbyte RAM banks. It includes seven other LCAs to implement switching and controlling functions. To compare the DEC-PERLE-1 CCM's performance with that of the software approach, we used a C program that executes the disjoint-sharp operation on two arrays of cubes. We also used this program on the CCM to solve all minterms with three, four, and five binary variables. We compiled the C program on a GNU C compiler v.2.7.2 and ran it on a Sun Ultra5 workstation with a 64-Mbyte RAM memory. For the CCM, we used a 1.33-MHz clock (with a 750-ns clock period). Table 2 (next page) shows the results of this experiment.

Table 2 (next page) shows that the software approach takes about one-fourth the time of the DEC-PERLE-1 CCM. But the Sun Ultra5

Table 2. Comparison of CCM prototypes with a software approach.

No. of variables	Execution time, Sun Ultra5 (μs)	Execution time, 1.33-MHz CCM time (μs)	Speedup for 1.33-MHz CCM (μs)	Execution time, 4-MHz CCM (μs)	Speedup for 4-MHz CCM (μs)
3	111	546 × 0.75 = 409.5	0.27	611 × 0.25 = 152.75	0.72
4	268	1,285 × 0.75 = 963.75	0.28	1,486 × 0.25 = 371.5	0.72
5	812	3,405 × 0.75 = 2,553.75	0.32	4,078 × 0.25 = 1,019.5	0.80

workstation's CPU clock is 270 MHz— 206 times faster than the CCM's clock. Therefore, although the CCM design is slower than the software implementation, we can still state that it is very efficient for cube calculus operations. Table 2 also shows that the more variables the input cubes have, the more efficient the CCM. This is because the software approach must iterate through one loop for each variable present in the input cubes. However, the clock period of 750 ns is too slow. The BIU state diagram shows that delays from an empty carry path and a counter carry path only occur in a few states. Thus, if we can give a little more time to these states—an easily achievable situation—we could speedup the clock of the entire CCM. For example, if state P2 of the BIU needs more time for the delay of counter carry path, we can add two more states in series between states P2 and P3. These two extra states do nothing but give the CCM two more clock periods to evaluate signal prel_res, which means that the CCM has three clock periods to evaluate signal prel_res in state P2 after adding two more delay states. After making similar modifications to all these states, the CCM can run with a 4-MHz clock frequency (clock period of 250 ns). Table 2 shows these results. It is difficult to increase the clock frequency again with this mapping because other paths, like memory paths, have delays greater than 150 ns. Table 2 illustrates a real need for human intelligence combined with the EDA tools to optimize the FPGA architectures.

From this comparison, we can conclude that it is not efficient to map a CCM design with a complex control unit and complex data path to a board such as the DEC-PERLE-1. Because our CCM mapping sends many signals through multiple FPGA chips, the signal delays are large. For instance, if we directly connect the memory banks and registers, the memory path has a delay of only 35 ns. But the DEC-PERLE-1 memory path has a delay of 160 ns.

Another issue is that the XC3090 FPGA is now outdated technology, so it's at a disadvantage when compared with more modern microprocessors. The latest FPGAs from Xilinx, Altera, or other vendors have more powerful CLBs and more routing resources, and greater speed because of deep-submicron processes. This direction in FPGA technology will continue, providing an advantage to their use in massively parallel accelerators.

For instance, if we map the entire CCM onto a single modern FPGA chip or a special connection pattern of modern FPGAs, we can speedup the CCM multiple times because of the following factors:

- The signals do not need to go through multiple chips, reducing routing delay.
- The new FPGA chips provide more powerful CLBs and routing resources, allowing denser CCM mapping. This also reduces the routing delays.
- Implementation of new FPGA chips in deep-submicron technology reduces CLB and routing wire delays. For example, CLB delay on an XC3090A is 4.5 ns, while delay on a Virtex II is 2.5 ns for a much more powerful cell.

We expect that with the new FPGAs and the corresponding FPGA-based boards, new versions of CCMs will become a true competitor to software—in terms of speed—for robotics applications.

To our knowledge, CCM is the first logic machine for MV logic, universal logic, and cube representation. It generalizes the previous machines of T. Sasao,[3] Ulug and Bowen, Zakrevskij, and others.[4]

The results of our experiments indicate that future CCM processors could provide significant speedups for many applications. The

CCM architecture is geometrically regular and scalable, an important advantage for its implementation with FPGAs. Designers can also implement CCM processors as VLSI ASIC chips by using a mostly fixed architecture and only small reprogrammable lookup tables. This implementation will result in much higher speedups. MICRO

References

1. Q. Chen, *Realization of a Universal Cube Calculus Machine in DECPeRLe-1 FPGA Emulator*, master's thesis, Dept. of Electrical and Computer Eng., Portland State Univ., Ore., 1998.
2. J. Vuillemin et al., "Programmable Active Memories: Reconfigurable Systems Come of Age," *IEEE Trans. VLSI Systems*, vol. 4, no. 1, 1996, pp. 56-69.
3. 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.
4. 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.

Marek Perkowski is a professor of Electrical and Computer Engineering in the ECE Department at Portland State University, Oregon. His research interests include multiple-valued logic, logic synthesis, testing and verification, reversible and quantum computing, intelligent robotics, walking robots, robot soccer, machine vision, and VLSI design. Perkowski received a PhD in automatic control from the Warsaw University of Technology, Poland. He is a member of the IEEE Computer Society and vice-chair of technical activities of its Technical Committee on Multi-Valued Logic. He is an editorial board member of the *Soft Computing Journal* and an academic advisor of the PSU student chapter of the IEEE Robotics and Automation Society.

David Foote works at Intel. His research interests include computer hardware and system design and networking. Foote received a BS in physics from Linfield College in McMinnville, Oregon, and an MS in electrical and computer engineering from Portland State University, Oregon.

Qihong Chen is a senior software engineer at the Portland Development Center of Oracle Corporation. His research interests include database systems, software engineering, software-hardware integration, and FPGA design. Chen received a BS in electronics and information technology from the Ocean University of Qingdao, China, and an MS in electrical and computer engineering from Portland State University, Oregon.

Anas Al-Rabadi is a PhD candidate at Portland State University. His research interests include logic synthesis, signal processing, image processing, regular structures, intelligent robotics, machine learning, system architectures, logic decomposition, reconstructability analysis, logic factorization, spectral methods, XOR logic, reversible logic, quantum logic, and logic testing. He is a member of the IEEE, the ACM, Eta Kappa Nu, Tau Beta Pi, Sigma Xi, and the Robotics and Automation Society of the IEEE.

Lech Jozwiak is an associate professor at the Faculty of Electrical Engineering of the Eindhoven University of Technology, Netherlands. His research interests include system and circuit theory and technology, design technology including EDA tools, artificial intelligence, multiple-valued logic, VLSI circuit and system synthesis, application-specific processors, embedded systems, programmable hardware, and reconfigurable computing. Jozwiak received a PhD in technological sciences (in the field of automatic control and computer engineering) from the Faculty of Electronics, Warsaw University of Technology, Poland. He is a member of the IEEE, the EDAA, and the Euromicro board of directors. He is cofounder and steering committee chairman of the Euromicro Symposium on Digital System Design, and an editor of the *Journal of Systems Architecture* and the *Journal of Computer Research*.

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.