An Associative Architecture for Genetic Algorithm-Based Machine Learning

Kirk Twardowski, Loral Federal Systems – Owego

Machine-based learning will eventually be applied to solve real-world problems. Here, an associative architecture teams with hybrid AI algorithms to solve a letter prediction problem with promising results.

Systems architects have continually sought to design machines with ever-greater levels of human-like autonomy and intelligence. It is widely recognized that the potential for such machines is nearly limitless, as evidenced by recent achievements involving autonomous agents, database mining, speech processing and translation, adaptive vision systems, visualization systems and animation. The results promise radical change in how we will eventually interact with our computers. Currently available systems, of course, are far from attaining real-world performance in such areas, largely due to a lack of computational power.

Researchers of massively parallel artificial intelligence seek to capitalize on advances in computer architecture to develop novel AI techniques that fully exploit the parallel capabilities of such powerful machines. The combination of AI and massively parallel computing will couple sophisticated knowledge-processing models with vast computational resources, which has the potential to eliminate the computational bottleneck that now prevents many AI systems from offering practical solutions to real-world problems.

This article describes an investigation and simulation of a massively parallel Learning Classifier System (LCS) that was developed from a specialized associative architecture joined with hybrid AI algorithms. The LCS algorithms were specifically invented to computationally match a massively parallel computer architecture, which was a special-purpose design to support the inferencing and learning components of the LCS. The LCS's computationally intensive functions include rule matching, parent selection, replacement selection, and, to a lesser degree, data structure manipulation.

Learning Classifier Systems

Learning Classifier Systems, introduced by Holland[1], are general-purpose machine learning systems designed to operate in uncertain, noisy environments that provide infrequent and often incomplete feedback. An example of such an environment might be a chemical plant, where an LCS would perform process control. An LCS comprises three layers: a parallel production system, a credit assignment algorithm, and classifier discovery algorithms. The production system models the problem domain as clusters of highly standardized rules called *classifiers*, and it provides

0018-9162/94/ $4.00 © 1994 IEEE

a basic match-select-act inferencing cycle with parallel-classifier activation. The credit assignment algorithm evaluates a strength for each classifier based on feedback from the environment. This strength serves as a measure of a classifier's utility to the LCS and is used both in the inferencing process and in the discovery of classifiers. Classifier discovery algorithms are typically a combination of genetic algorithms and several heuristic methods. Together, credit assignment and classifier discovery are the techniques that endow the LCS with its adaptive capability, which is what enables machine learning systems to respond to changing conditions in a problem domain.

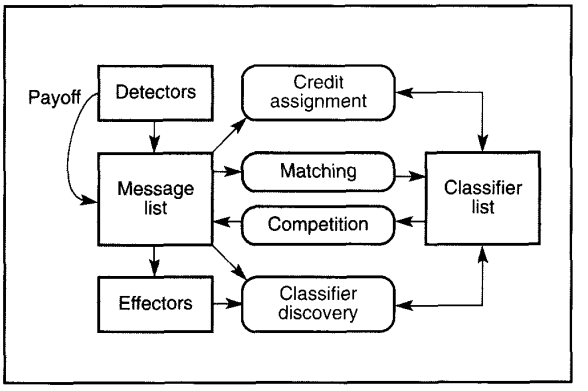


Figure 1. Block diagram of the Learning Classifier System components. The screened components compose the production system layer.

Rule-based production system. The LCS production system layer bears many similarities to rule-based expert systems. In particular, the production system's knowledge is encoded in a set of classifiers processed by a cyclic match-select-act inferencing algorithm. The primary difference between the two system types lies in the production system's mechanisms for simultaneous classifier activation, which makes it a parallel-classifier-based system. On the other hand, expert systems are sequential in nature, permitting only one rule to be processed at a time. Short-term working memory is maintained on a global *message list* that stores internally generated messages as well as input and output environment communication messages. A set of detectors and effectors provides the message-based interface to the environment. An example of a detector is a temperature sensor, whereas an example of an effector is a robotic arm or a valve.

Each classifier has a simple IF *condition(s)*, THEN *action* syntax (for example, IF temperature is greater than 100°, THEN open valve). Conditions and actions are fixed-length strings and are typically identical in length for all classifiers. The symbol alphabet used to compose both the condition and action strings is {0, 1, #}. The # symbol represents a don't-care character that can match either 0 or 1. Messages are identical in structure to conditions and actions, except they contain no # symbols.

An LCS production system, therefore, consists of a classifier list, a message list, a set of detectors, a set of effectors, and a feedback mechanism (see Figure 1). Also shown are the credit assignment and classifier discovery components (layers). The basic execution loop governing the interactions between these components consists of six steps in a single execution cycle:

Glossary

Bias — Many of the decisions made in the Learning Classifier System are of a stochastic nature. They are controlled by the bias, which is a numeric value stored with each individual classifier in the LCS.
Bid — A fractional amount of strength paid by a classifier for the right to post a message that is used in the bucket brigade algorithm.
Classifier — A basic component of knowledge representation in an LCS that is analogous to a rule in expert or production systems.
Classifier discovery — That part of the system that uses heuristics, most notably the genetic algorithm, to explore new concepts by creating new classifiers.
Competition — A process, which is based on a classifier's strength, that decides which classifiers are granted access to limited system resources (that is, the message list).
Crossover — A basic operator in the genetic algorithm that generates a new classifier from subsections of parent classifiers.
Detectors — Sensors that translate environment conditions into the messages processed by the LCS.
Effectors — Environment manipulators used by the LCS to perform actions.
Fitness — A relative measure of a classifier's utility to the LCS in solving a given problem.
Genetic algorithm — A search-and-optimization algorithm based on the mechanics of biological evolution.
Payment — The strength value transferred between two classifiers within the bucket brigade algorithm. Payment is made to the classifier that generated a message from the classifier that matches the message.
Payoff — The scalar reinforcement value received from the environment as a form of reward or punishment.
Spatial locality — The physical distribution of classifiers within the array of processing elements where parents and replacement classifiers are selected such that they are physically colocated.
Specificity — A measure of the number of different messages that can match a classifier. A classifier can match from one to hundreds of messages that are either internally generated by the LCS or issued from the environment. Classifiers that are very general match many messages and therefore handle default conditions. Classifiers that are very specific match few messages and therefore handle special cases in the environment.
Strength — Numeric estimate of fitness that controls many aspects of a classifier's behavior in the LCS, that is, in the competition to post new messages and its probability of being selected as a parent or a replacement.

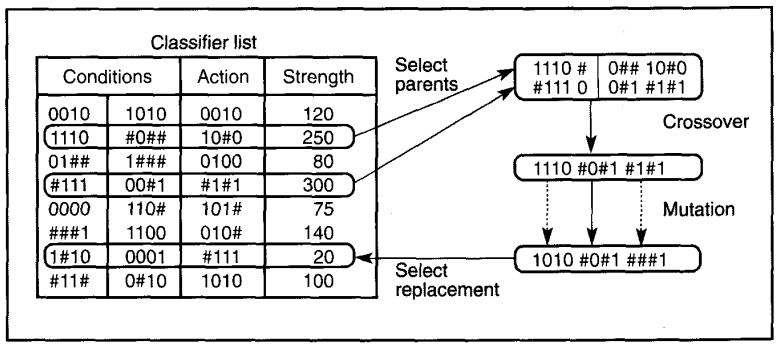


Figure 2. Example of a genetic algorithm cycle.

(1) any messages from the environment detectors are added to the current message list, (2) the contents of the message list are matched against all the conditions of all the classifiers, (3) those classifiers whose conditions were matched compete for the right to post messages to the message list such that those with greater strength are favored to win, (4) the winners of the competition create new messages based upon their actions and the matching messages, (5) the new messages are added to the message list, and (6) the effectors perform any actions specified in the message list.

Credit assignment. Credit assignment has long been recognized as a difficult problem inherent in any learning system composed of many interacting components (for example, classifiers) that contribute, over time, to the overall performance. The purpose of credit assignment in an LCS is to distribute feedback from the environment in the form of a scalar reinforcement value such that beneficial classifiers are rewarded and detrimental classifiers are penalized with respect to the desired outcomes.

Holland's[1] proposed *bucket brigade* algorithm is a mechanism that can potentially solve the credit assignment problem in an LCS. The objective of the bucket brigade algorithm is to distribute payoffs received from the environment to the appropriate classifiers in the form of strength adjustments. When the environment determines that the LCS has acted in a beneficial way (for example, correctly regulates temperature in controlling a process), it rewards (pays off) the system in terms of added strength. Conversely, if the LCS has acted in a harmful way, the environment penalizes it by taking strength away. This is important because these adjustments shape the adaptive (learning) ability of the LCS: Classifiers whose strength has been increased are more likely to be selected when a similar problem next needs to be solved, while those whose strength has been diminished are less likely to be selected.

As the term bucket brigade implies, strength is taken in small quantities from those classifiers that lead directly to payoff (active when payoff is received) and given to those classifiers that lead indirectly to payoff ("stage-setting" classifiers). Conceptually, the bucket brigade algorithm operates on chains of classifiers in which strength is being passed backward from the payoff-receiving classifier to previously active classifiers. The algorithm consists of two steps for each posting classifier: (1) reduce the classifier's strength by an amount equal to a fraction (approximately 1/10) of its strength, and (2) distribute this amount among classifiers that generated, in the previous time-step, the messages that satisfied this classifier. Classifiers posting effector-actuating messages when payoff is received share the payoff amount, and have their strengths updated accordingly.

Classifier discovery algorithms. While the bucket brigade is an effective mechanism for the temporal aspects of credit assignment, it cannot modify the system's knowledge structure. The ability to modify the system's internal knowledge structures is crucial for an LCS to learn new behaviors or adapt to a changing domain. What is needed is the ability to create new classifiers and delete those that have proven to be of little value.

The primary classifier discovery mechanism in an LCS is the *genetic algorithm*,[2] which is why a simplistic string representation is used for classifiers. The genetic algorithm is a heuristic search procedure modeled on natural evolution in an attempt to capture evolution's adaptive and optimizing features in a practical algorithmic form.

In an LCS, the genetic algorithm is periodically invoked to create new classifiers. The algorithm's basic execution cycle is:

(1) from the classifier list, randomly select pairs of parent classifiers such that higher-strength classifiers have a greater chance of selection,
(2) create new classifiers by applying genetic operators to the parents, and
(3) randomly select those classifiers to be replaced by the newly generated classifiers such that lower-strength classifiers have a greater chance of selection.

In the prototypical genetic algorithm, there are two genetic operators: *crossover* and *mutation*, which are applied to the selected parent classifiers to create new classifiers. To form a new classifier, the crossover operator pieces together sections from two parents, while the mutation operator, with a very low probability, alters randomly selected bits within a classifier.

Figure 2 shows a single genetic algorithm cycle that has been applied on classifiers with two 4-bit conditions. For emphasis, selection of parent and replacement classifiers is shown as a maximum or minimum function, respectively. Crossover occurs between the fifth and sixth bits, while bits 2 and 10 are mutated.

The associative architecture

There were two key reasons compelling the choice of a specialized associative architecture: (1) searching occurs frequently during LCS functions (rule matching, parent selection, replacement selection, and data structure manipulation), and (2) the independent nature of the individual classifiers made them well suited to the SIMD (single instruction, multiple data) paradigm of associative computing. For these reasons, we believed a computationally efficient implementation was well worth investigation.

To date, two notable parallel LCSs include Robertson's[3] *CFS on the Connection Machine and Dorigo's[4] Alecsys, which runs on an array of transputers. Of these, *CFS is most similar to the approach described here because it is a SIMD massively parallel system. Neither *CFS nor Alecsys, however, incorporates

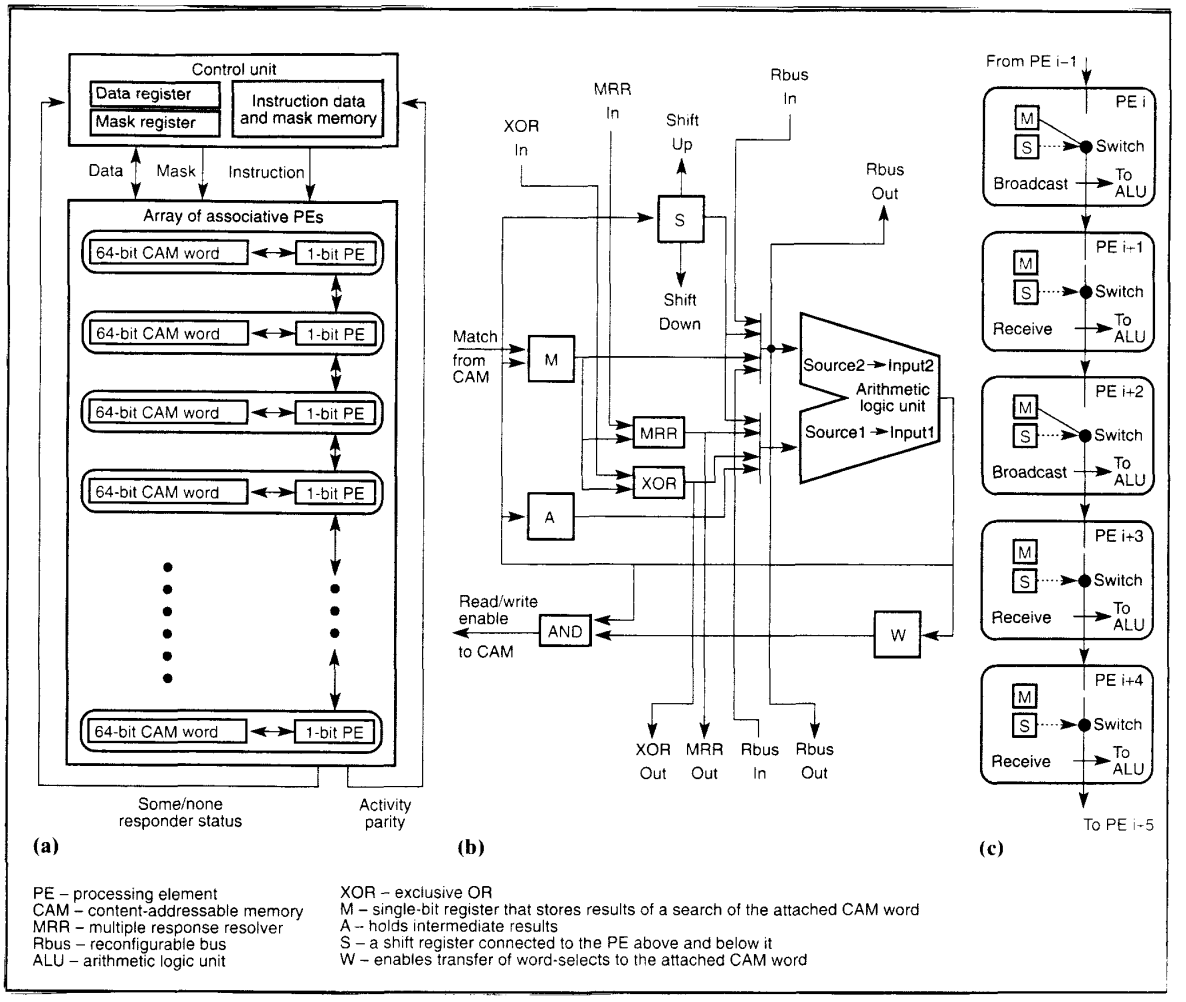


Figure 3. Three views of associative architecture: (a) high-level generalized block diagram; (b) processing element logic diagram showing the four single-bit registers: *M* stores results of a search of the attached CAM word; *W* enables transfer of word-selects to the attached CAM word; *S* is a shift register connected to the PE above and below it; and *A* holds intermediate results; (c) reconfigurable bus operation.

a parallel GA model as does our implementation as described later. A parallel genetic algorithm is important for two reasons: (1) it extracts as much parallelism from the algorithms as possible, and (2) it improves system performance with respect to the number of classifiers. Accurate execution times are not available for either system, so a meaningful performance comparison will not be possible until further research is conducted.

The architecture is a linear array of fully associative processing elements that consist of 64 bits of content-addressable memory, coupled with a 1-bit row processor to provide response processing, activity control, multiple response resolution logic, and inter-PE communication. Memory and PE size determination was based on commercially available CAM chips or on those in development, as described in the literature[5] and by Stormon during the "Associative Processing and Applications Workshop" presented at Syracuse University in 1992.

Figure 3a shows a high-level view of the architecture. The array of PEs operates in a SIMD mode and therefore has a controller that is responsible for generating and broadcasting instructions and data to the array, as well as accumulating and testing global feedback information. The controller contains a data register, which holds the data broadcast to the array, and a mask register that determines which bit columns of the array are active during writes and matches. This architecture is an example of traditional, fully parallel associative processing, and it provides essential associative computing capabilities, such as

- fully parallel search of all memory,
- constant time responder/no responder status,
- multiple response resolution to select a single processor from many,
- efficient broadcast of data and instructions from controller to array, and
- efficient one-to-one data transfer between processing elements and the control unit.[6]

In addition, the architecture provides an extended communication capability in the form of a reconfigurable bus similar to those found in many of the more re-

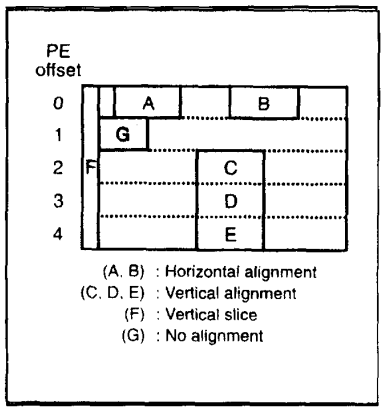


Figure 4. Examples of the various alignment cases possible when variables are mapped onto processing elements.

cent VLSI implementations of associative processors.[7-9]

Processing element. Figure 3b is a detailed diagram of the 1-bit processor within each PE. There are four single-bit registers used for dedicated functions, temporary results storage, or both. The M register stores the results from a search of the attached CAM word. The W register, since it enables the transfer of the word-selects to the attached CAM word, effectively controls local activity (that is, whether the PE executes instructions broadcast to it). The S register is a shift register connected to the PE directly above and below. The A register primarily holds intermediate data. The ALU (arithmetic logic unit) can calculate any function of two inputs and can be loaded into any of the four registers.

The multiple response resolver (MRR) behaves like a priority circuit. Its output is a single bit that corresponds to the topmost active bit in the M register. The MRR resolves the situation that results when multiple PEs, which need to be processed individually, respond to a match pattern.

Output of the M register feeds one input of an XOR (exclusive-OR) gate, with the other input being the output of the XOR in the PE directly above, thus forming a chain of XOR gates that connects all PEs. The XOR chain has two functions: to enumerate the active responders and quickly count responders.

In addition to the shift register, a reconfigurable bus (Rbus) lets the PEs be connected as arbitrary contiguous segments. For operations such as a parallel-prefix add, a more significant performance gain can be realized through the Rbus, which is more effective for long-distance communication between PEs than for simple shifts of data. Communication on the Rbus is unidirectional and occurs in either a downward or upward direction. Each segment starts at a broadcasting PE and continues to the next broadcasting PE, where the S register controls the connectivity, as shown in Figure 3c. It is important to note that Figure 3c is a logical, not physical, representation of the design.

Instruction set. The instruction set allows the simultaneous execution of three different operation types — array, shift, and ALU. Within each PE, the read, write, and match instructions control the operation of the CAM word. At the locations activated by the word-select lines, read returns data and write modifies the contents of the CAM array. The data register stores data that is written, and the mask register's contents determine the bit columns to be modified. The match instruction determines those locations in the CAM array that match the value in the data register. The bit columns to be searched are specified by the mask register; therefore, individual bits or subfields within the array can be isolated for a search. The shift and ALU operations control the S register and the ALU outputs, respectively. The shift operation results in an unconditional change of the S register in all PEs.

Programming model. The programming model typically employed in fully parallel associative architectures is often called *data parallel* and is the same as that found on many of the bit-serial massively parallel machines such as the DAP, Thinking Machine's Connection Machine CM-1, and the MasPar MP-1. In the data-parallel model, there is a copy of each parallel variable in every PE within the array; thus, if a machine contains 8,192 PEs, there will be 8,192 copies of each parallel variable. In fully parallel VLSI implementations of CAM, however, the length of the CAM word can be a limiting factor. While a single CAM word appears to be adequate for image processing tasks,[9,10] CAM word length severely limits most other kinds of processing that require more PE memory. In these instances, a logical-to-physical mapping is necessary to allocate a set of PEs to each set of variables being processed in parallel.

In the programming model selected for the LCS design, a contiguous set of physical PEs is allocated as a logical PE that processes a record. *Record* refers to a collection of data-parallel variables to be processed by a single logical processor. This set of PEs, acting as a single processor, then processes the data within that record. This model is in direct contrast with, for example, the C* Connection Machine programming language, where a single physical PE can support as many virtual PEs as will fit within available memory. In many cases with our LCS model, there is a loss of parallelism as only one of N physical PEs within a logical PE performs useful work at any given time. Occasionally, however, it is possible to exploit parallelism within a record so that more than one physical PE per logical PE is active.

The variables within each record consist of a contiguous set of bits within a single PE. Unlike a conventional computer that can use a single address parameter to identify and locate a variable, the associative processor under our programming model requires three parameters:

- the starting bit position of the variable with a PE,
- the variable's length in bits, and
- the offset of the PE containing the variable.

The starting bit position is analogous to the address in a conventional machine. A variable requires a length because there are no predetermined lengths for variables; a variable can be anywhere from 1 bit long to as long as, or longer than, the entire CAM word. The offset identifies the PE containing this variable out of all physical PEs that constitute the logical record.

Mapping data onto CAM. The memory organization of the logical PE is a two-dimensional array of bits with one dimension being the physical PE offset and the other being the starting bit position. It is essential therefore to consider the alignment relationships between a record's variables when they are mapped onto the PEs. These relationships determine the amount of parallelism that can be extracted from the array. The alignment relationships, depicted in Figure 4, can be classified as horizontal alignment, vertical alignment, no alignment, or vertical slice.

Horizontal alignment applies to items

Table 1. Execution time of primitive operations.

Name	Function	Mode: Scalar	Mode: Vector	Mode: Segmented
Increment	$x_1 = x_1 + 1$	$3 + 3m_1$		
Decrement	$x_1 = x_1 - 1$	$3 + 3m_1$		
Add	$x_1 = x_1 + x_2$	$7 + 4m_1$	$7 + d_{12} + m_1(4 + d_{12})$	
Subtract	$x_1 = x_1 - x_2$	$7 + 4m_1$	$7 + d_{12} + m_1(4 + d_{12})$	
Multiply	$x_3 = x_1 * x_2$	$7 + f_1 + (m_{con}/2)$ $(2 + m_1(4 + d_{12}))$	$5 + f_1 + 2d_{12}$ $+m_2(9 + 2d_{13} + f_2 + m_1(4 + d_{12}))$	
Divide	$x_3 = x_1 / x_2$	$9 + f_1 + 2d_{13} + m_3$ $(10 + f_1 + d_{13} + 6m_1)$	$9 + f_3 + d_{13}$ $+m_3(15 + f_1 + d_{13} + d_{12} + m_1(7 + d_{12}))$	
Reduce Add	$x = \Sigma_{j=1}^{i} x_j$	$m_i + (2 + \lg(N))$		
Scan Add	$x_i = \Sigma_{j=1}^{i} x_j$		$3 + \lg(N)(9 + 2d_1 + 5m)$	
Shift			$3 + m_1(3 + d_{12})$	
Compare		$7 + f_1 + 1.5m_1$	$10 + 2f_1 + m_1(4 + d_{12})$	
Minimum		$1 + 2m_1$		$18 + 4f_1 + 7m_1$
Maximum		$1 + 2m_1$		
Move Field			$3 + m_1(3 + d_{12})$	
Count		$\lg N$		
Enumerate			$4 + 2\lg(N)$	
Random			$9 + f_1 + 4m_1$	
Send Field			$15 + f_1 + f_2 + 2m_1$	
Spread				$17 + 5f_1 + 5m_1$

Notes:

m_i length, in bits, of operand

d_{ij} distance between PEs holding operands x_i and x_j

f_i distance between operand x_i and start of record

N number of active PEs

that must be stored in the same PE. For example, the destination and source operands of a multiply operation should be within the same word to minimize inter-PE communication overhead. *Vertical alignment* specifies that two items stored in different PEs are to be aligned so that they both start at the same bit position. The conditions of each classifier are an example of this relationship; storing them in a vertically aligned manner means both can be matched simultaneously against messages. Vertical alignment exemplifies parallelism between record variables. *No alignment* is suitable for those items that have no interdependencies and can be placed anywhere within the allocated PEs. A *vertical slice* is a single-bit column that extends the entire length of the PE array and is an exception to the programming model introduced above since it consumes a bit at every PE. Vertical slices typically provide storage for maintenance purposes or for temporary storage of a PE's register contents. A vertical slice can also hold data that is processed in a bit-parallel manner by all the ALUs.

Associative primitives. Implementing the LCS algorithms requires a core set of arithmetic, logic, and communication primitives. These algorithms are inherently bit serial, since the PE is only a single bit wide. Consequently, operations can take many more computation cycles to complete than with bit-parallel algorithms. However, since many operations are performed simultaneously, the increase in cycles is amortized over the total number of results generated, giving a superior throughput. This does assume that the parallelism is great enough to sufficiently amortize the cost. Furthermore, since the architecture lets operands be any length, efficiency gains are often achieved at the expense of precision, which suits our purposes in the model.

Table 1 lists the primitive operations used by the LCS algorithms, and Table 2 describes the higher-level primitives. Each column of Table 1 shows the execution time in machine cycles, for each of three possible execution modes. The *scalar* mode applies when the controller broadcasts a scalar value to the active set of PEs. *Vector* mode occurs when all operands are contained within the PE array. *Segmented* mode supports the execution of segmented scans and reduction primitives[11] as well as long-range communication via the Rbus.

As is evident in Table 1, some of the operations (for example, scan add in segmented mode) were not implemented, primarily because the LCS algorithms did not require them. The architecture, however, has no limitations that would prohibit the rest of the operations from being developed.

Execution time parameters have two

Table 2. Higher-level associative primitives.

Name	Description
Count	Using the XOR logic, assemble a count of the number of active PEs.
Enumerate	Assign consecutive numbers to the active PEs via the XOR logic.
Random	Generate a random number in each active PE, via a one-dimensional cellular automata algorithm.
Send Field	An Rbus communication primitive to transmit fields between specially marked PEs. Restricted to transmitting between nonoverlapping pairs of PEs due to the nature of the single-wire bus connection between PEs.
Spread	A segmented broadcast from one PE to a set of physically adjacent PEs as controlled by a bit vector that establishes how the array is broken into segments.
Segmented Minimum	Find the minimum value in each segment of the array, where the segmentation is controlled by a bit vector contained in one of the PE registers.
Scan Add	Tabulate a running sum over all the currently active PEs. Scan Add uses the Enumerate primitive to control the connectivity on the Rbus.

dimensions: m_i represents the length of operand i, and d_{ij} represents the distance between the PEs containing operands i and j. As an example, consider an add instruction that adds two variables and stores the result in the first variable. If these two variables are located on different PEs, the contents of the second variable must be bit-serially shifted to the first as the add progresses. Thus, if the operands are m bits long, $m \times d$ cycles will then be required in addition to the four cycles needed to read the operand bits, calculate the new data and carry bits, and update the CAM word. The additional $7 + d$ cycles are mainly "cleanup" code for overflow and underflow cases.

Associative implementation of LCS

All three LCS layers were implemented with the primitive operations just described. Next, we examine the mapping of program data structures onto the CAM and how the primitive operations were applied.

CAM data structure. Our LCS contains two primary data structures: the message list and the classifier list. There are three ways to map them onto the associative processor — store messages in the PE array (message-parallel), store classifiers in the PE array (classifier-parallel), or store both in the PE array (jointly parallel). This specific LCS implementation is based on the classifier-parallel approach for two reasons: (1) it minimizes transferring messages between the array and the controller, and (2) technology already exists to support 1,000 to 10,000 classifiers in a design that could be easily adapted for a desktop PC application, as explained by Stormon at the Syracuse University workshop in 1992.

Record size considerations. In addition to conditions, action, strength, and specificity, each classifier requires a number of flags and temporary storage; all the variables that compose a classifier are allocated to a single record. To attain the approximately 280 bits of memory required by a classifier record, a minimum of five PEs (320 bits) must be allocated per record. Figure 5 shows the memory map for the classifier record that was used for the simulation experiments. The minimum number of PEs has been allocated to each record to maximize the number of classifiers that can be supported.

Figure 5 identifies the record variables that are statically defined for the dura-

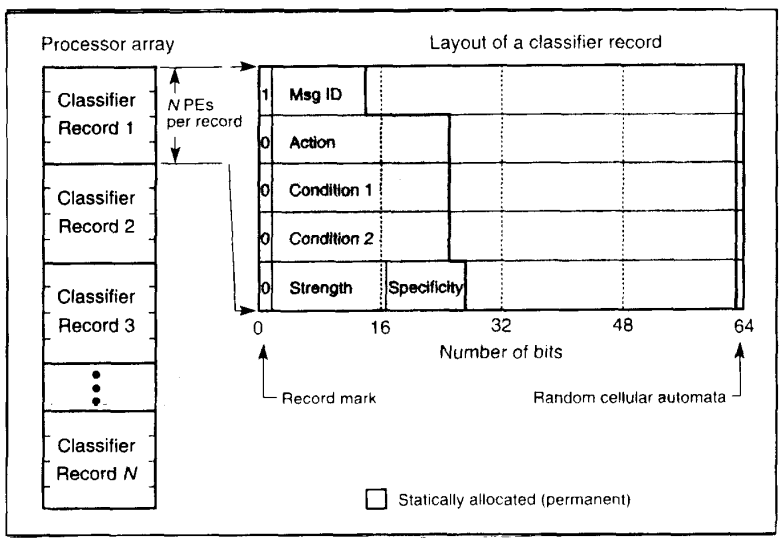


Figure 5. Memory map layout of the static variables in the processing elements.

tion of the program. Note that conditions 1 and 2 are vertically aligned to speed up message matching, since both can be compared simultaneously. Remaining space is allocated for temporary storage as needed. The first bit of all five PEs is the record mark that identifies the start of each record. Next to the record mark are the action, conditions, and strength. The first PE has a message identification variable that is reserved for linking classifiers with the messages they posted. The last bit of each PE holds the state of a cellular automaton that generates random numbers.

Condition and action representation. The fully parallel associative architecture, due to its capability to selectively mask search bits, has the ability to store a don't care (#). The don't care will match either a one or a zero in the search pattern. This is particularly useful for representing the conditions and action of a classifier, which use the # symbol in just that manner. Each condition and action symbol uses two CAM bits, where a $0 \equiv 01$, $1 \equiv 10$ and $\# \equiv 11$. The search patterns are #1 for a zero and 1# for a one: Both of these match the 11 used to represent the # in conditions and action, as well as their respective symbol.

The production system layer. The core processing loop within the LCS is the five-step match-select-act process listed in Table 3: (1) match classifiers, (2) create messages, (3) post new messages, (4) extract messages, and (5) process effectors. The continual repetition of these five steps constitutes the largest portion of the computational effort. Note that this processing loop differs in two respects from the earlier processing loop description: (1) the add detector messages step has been disregarded since this doesn't involve the array, and (2) the order of the create message and post message steps has been reversed to simplify the parallel implementation.

Match classifiers. A special-purpose associative architecture was selected for the LCS largely due to the matching requirements of the match classifier step. The CAM-based design reduces the runtime of this step virtually to a constant, regardless of the number of classifiers. Moreover, the associative organization means that match status can be maintained without pointers or intermediate structures. Unlike associative processing, sequential processing would, in order to reduce runtime, need to establish a linked list of candidate classifiers, each with its own list of matching messages. The associative architecture avoids this situation by using status flags that can be matched in parallel or, since the match cost is very low, by reprocessing the message list. Our LCS features both techniques.

The message list is processed in two passes. In the first, all candidate classifiers are determined. During the second pass, a copy of the matching message is stored at each candidate classifier and marked as "used." Each candidate classifier matching an internal message — one posted by a classifier on the previous cycle — has a match count incremented. The message stored with the candidate classifiers is used for the *create messages* step, and the credit assignment layer later uses this match count to determine strength-payments distribution to classifiers active in the previous time-step.

Create messages. The action component of each classifier is the template for new-message construction. Recall that the symbols in the action are from the set: {0, 1, #}. The 0, 1 is copied directly to the new message, whereas the # is a "pass-thru" token that accepts the corresponding bit from a matching message. This algorithm is similar to a field-move operation, except that it conditionally copies bits from the source field. As implemented, new-message creation moves only 0s and 1s from the action variable to the message variable containing the message stored during the match step. The #'s found in the action are not copied into the message variable, which lets the matching message define the new message at these bits.

Post new messages. The message list is a constrained resource in the system as it has space enough for only a limited number of messages. Furthermore, there is a limit to the number of each message type permitted on the list. Consequently, the primary task of message posting is to count the number of new messages. If there are too many of the given type, then the system runs a competition to determine those that will actually be posted. Another task performed during this step is bid calculation for each prospective message. The bid is used to bias the competition and is stored with each message for reference by the bucket brigade algorithm. Typically, the bid is a function of strength and specificity; in our LCS implementation it is strength times specificity.

The competition operation conducts a parallelized random selection by first performing a scan-add of the bids and, for each message to be selected, generating a random number between zero and the

Table 3. Core processing loop in the production system layer.

Step	Name	Description
1	Match classifiers	Each message is matched against all classifier conditions. Each classifier with all conditions matched becomes active.
2	Create messages	Each active classifier, based on its actions and a matching message, creates a new candidate message for posting to the message list.
3	Post new messages	If required, a competition is run to see which messages are posted to the message list; if not, all messages are posted to the message list.
4	Extract messages	Messages to be posted are read from the array and loaded into the message list in controller memory. A tag is associated with each classifier/message pair for use in the credit assignment layer.
5	Process effectors	The message list is processed by the effectors, and messages are consumed by any effector that they match.

sum of all the bids. Each random probe searches the array to find the classifiers whose bid is greater than that probe. The MRR then selects the topmost classifier as the winner for this step.

Extract messages. Message extraction requires three passes through all new messages. The first pass assigns each message and its generating classifier a tag that links them together for the bucket brigade algorithm in the credit assignment layer. Each message identifies its generating classifier from the tag via a single match instruction. The 8-bit tag variable is incremented whenever a message is generated, with the assumption that fewer than 128 messages are created during each cycle of the production system layer. The remaining passes read the messages and bids, then insert them in the message list.

Process effectors. Effector processing is an inherently sequential operation that loops through the message list to see if any messages satisfy an effector. If so, that effector performs its function, and the respective message is removed from the list.

The credit assignment layer. The bucket brigade algorithm is the sole function performed by this layer, and its operation is driven by the contents of the message list. Each message specifies a transfer of strength to the classifier that posted the message from the classifiers it matched. Also at this time, the bid is deducted from the classifier that generated the message.

First, each classifier calculates a payment value; this is the bid divided by the number of internal messages it matched, because an equal share of the bid is paid to each message-generating classifier. Next, each internally generated message is processed sequentially. The message is first matched against the active classifiers to find those from which a payment is to be collected. Next, a reduce-add primitive calculates the total payment owed to the classifier that generated the message. Finally, the classifiers are searched again, this time with the message tag, to locate the generating classifier and store the payment it has received. After all messages are processed, all classifiers receiving a payment from a message have their strengths simultaneously updated with an add-vector variable primitive.

The associative search function of the array simplifies the execution of this algorithm by allowing a low overhead mechanism to quickly identify links between messages and classifiers. A sequential machine, on the other hand, would have to maintain a number of lists that link classifiers with messages and messages with their posting classifiers.

Classifier discovery layer. The classifier discovery layer is the most complex of the three LCS layers and uses a genetic algorithm as the discovery heuristic. There are nine steps involved that make heavy use of the communication bus as well as numerous other processor capabilities. It is worth noting here that our LCS implementation replaces the standard genetic algorithm with a parallel GA.[12] Parallel GAs employ the characteristics of parallel computers to build a model that more closely resembles natural evolution by introducing the concept of *spatial locality*. The standard GA selects parents and replacements from the entire pool of strings without any bias other than the weighted selection process. This is not, however, a realistic model of how evolution actually occurs. In reality, parents are most likely to reside within close proximity of one another. By limiting the distance between parents and the string their offspring will replace, a parallel computer becomes the logical choice to implement the parallel GA because of greatly reduced communication costs inherent in the architecture. Moreover, algorithm processing improves twofold. First, as expected, parallel processing increases the algorithm's execution speed. Second, a more subtle improvement results from the spatial relationships between the population members, which has the effect of allowing small pockets of the population to evolve somewhat independently from the rest. Consequently, as each subpopulation searches a different area of the solution space, a larger area of the solution space is searched simultaneously.

Parallel genetic algorithms build a model that more closely resembles natural evolution by introducing the concept of spatial locality.

Mark eligible parents. This step globally searches various classifier tags and numeric values to mark those classifiers that can be considered as potential parents.

Calculate fitness. A biased version of strength, called fitness, is used during parent selection. The bias increases the chance that those classifiers with higher strength will be selected. Fitness is normally calculated by raising strength to a prespecified power. These LCS simulations, however, simply set fitness equal to strength.

Select parents. The same parallelized random selection algorithm that was used in the competition to post messages is applied to parent selection, with the exception that the algorithm is now based on the fitness value just calculated. The number of parents selected is twice the number of classifiers to be generated, which is a fixed percentage of the total number of classifiers.

Implicit in a sequential genetic algorithm is the grouping of parents together for applying the crossover operator. In a fine-grained parallel genetic algorithm, this is problematic as it introduces the need for the classifiers to establish pairwise groupings. One parent of each group must then send a copy of its conditions, actions, and strength to its "mate." However, the reconfigurable capability of the Rbus suggests a method of grouping parents that maximizes bus utilization and is computationally less demanding. All parents are labeled as either even or odd depending on their location in the array, with the topmost parent being even. Each even parent is grouped with the odd parent immediately below it. Grouping the parents in this fashion is important as it splits the array into spatially disjoint segments that can make use of the Rbus without contention. Thus, all "even" parents can simultaneously broadcast to their "odd" mates via the Rbus, using the send-field primitive.

Send parents. Offspring generation by means of the crossover operator requires the conditions, actions, and strengths of the two classifiers. The send-field primitive supports this communication based

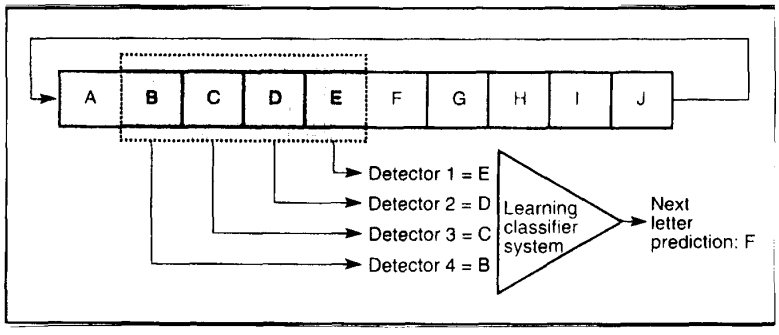


Figure 6. Letter sequence prediction problem domain.

on the parent grouping just described. In order to minimize communication time during offspring generation, vertical alignment is set up between the respective classifier components that will become offspring after the following step (create offspring). All even parents are then disabled; further processing occurs at the odd parent.

Create offspring. Offspring creation proceeds by first applying the crossover operator on selected parents at the odd mate location and then applying the mutation operator on the offspring thus generated. Both steps extensively use random number generation to determine the outcome of many decisions that are part of these steps. Decisions include determining

- which offspring are to be created by crossover as opposed to just copying,
- crossover point locations,
- the type of crossover to perform, and
- the number and location of mutations.

Crossover is similar to message creation since one variable is being conditionally copied into another (that is, the odd parent into the new offspring). The *copy state*, initially set to "no copy," controls the conditional copying of the odd parent. Crossover proceeds bit by bit through the entire classifier. The copy state is updated prior to the generation of each bit of the offspring, such that all PEs whose first crossover point matches the current bit set their copy state to "copy." All PEs whose second crossover point matches the current bit set their copy state to "no copy." Thus, for the range of bits between the two crossover points, the offspring originates from the odd parent.

Mutation changes up to three bits in each offspring, and for each one of these possible mutations it maintains a *mutation position* variable and a *mutation-active* flag. Like crossover, mutation proceeds bit by bit over the entire classifier, but now, when the current bit matches a mutation position in a classifier that has the respective mutation-active flag set, that classifier undergoes a mutation at this bit position.

The final step of creating the offspring is to calculate the new strength for the offspring. In this implementation, an average of the parent strengths is applied.

Duplication check. There is nothing to prevent the offspring generation step from producing many identical classifiers. In particular, high-strength classifiers have a tendency to reproduce rapidly, quickly dominating the entire set of classifiers and degrading system performance. A duplication check limits the number of duplicates by reading each offspring from the array and comparing it with the current classifier list. If the number of responders is greater than permitted, the offspring is eliminated.

Select replacements. Replacement selection relies on the segmented minimum primitive to build a local neighborhood around each offspring. From this neighborhood, a classifier is selected that will be replaced by the offspring. The size of the neighborhood, N, is typically a small integer. In our LCS simulations it was three.

Use of the segmented primitives requires that the segment boundaries be set up beforehand. Segment boundaries are created with a two-step process in which first the high, and then the low, segment boundaries are propagated outward from each offspring. Taken together, the upper and lower segment bounds demarcate the neighborhood of the offspring from which the replacement will be selected. If two offspring are within N of each other, their neighborhoods are

Table 4. Average number of active processing elements per call by primitive operation.

Primitive	Avg. cycles	Percent total	Number of classifiers 200	400	600	800	1,000	1,200
Subtract	118	19.94	24.1	59.6	101.8	165.7	198.1	268.1
Multiply-vector	957	17.49	28.6	44.0	69.2	27.4	46.1	38.6
Scan Add	992	14.15	47.1	87.1	120.4	229.7	203.3	210.3
Compare	87	13.48	46.2	113.2	177.6	391.0	416.4	559.9
Move Field	67	7.51	19.9	54.8	69.6	84.5	111.0	117.8
Add-vector	78	1.46	5.5	3.5	4.3	3.5	4.3	4.47
Reduce Add	119	1.24	200.0	400.0	540.0	688.0	998.6	1198.9
Random	68	0.95	...	...	...	...	...	...
Maximum	54	0.79	47.1	87.2	120.4	229.7	203.3	210.3
Decrement	59	0.55	200.0	400.0	600.0	800.0	1,000.0	1,200.0

merged and one of the two will be discarded, depending on the location of the replacement. If the two offspring are within $2N$, but further than N, then the lower bound of the topmost bound is shortened so it doesn't overlap that of the second. The segmented minimum primitive then selects the lowest strength classifier within each segment, which is then marked for replacement.

Send offspring. Following the replacements selection, the spread primitive broadcasts the offspring to the replacement.

Calculate specificity. The specificity for the offspring is calculated at its new location to minimize communication costs. Since specificity is just a count of the number of 0s and 1s in the conditions and actions of the offspring, the desired result is obtained by performing bit-by-bit compares and incrementing the specificity variable for each.

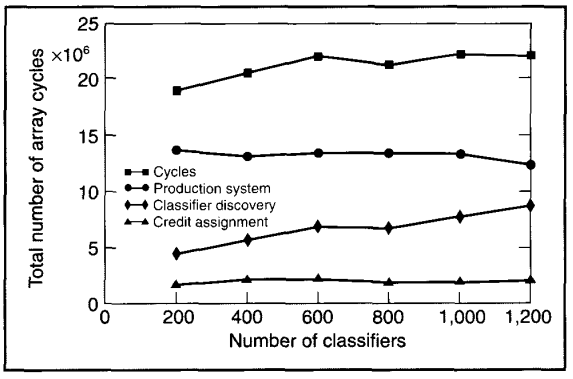


Figure 7. Total number of execution cycles for a varying number of classifiers, showing the number of cycles contributed by each of the three layers: production system, credit assignment, and classifier discovery.

Performance evaluation

An associative architecture simulator was developed on a MasPar MP-1 system that consisted of 8,192 4-bit processors. The simulator served as a highly instrumented testbed on which the performance of various algorithms was investigated. Additionally, the use of the MP-1 parallel computer with an architecture closely matched to that of the simulated machine proved highly effective in reducing the runtime of the simulations.

The LCS algorithms were exercised on this simulator and tested on a letter prediction problem. In this problem, the LCS detectors were a sliding window over a continually repeated sequence of letters, and the desired output of the LCS was a prediction of the next letter to become visible in the window, as shown in Figure 6. This was a difficult problem as the system had no knowledge of the problem domain to begin with, and had no meanings associated with its detector inputs or effector outputs. From a qualitative reinforcement signal that merely indicated "right" or "wrong," the LCS had to create a set of prediction rules.

Many simulations, which varied the number of classifiers from 200 to 1,200, were performed to test the effect on execution time. The number of processing elements involved was five times the number of classifiers, or from 1,000 to 6,000. Figure 7 shows the total number of machine cycles required to complete a simulation run of 4,000 cycles. In general, execution time increased slightly with respect to the number of classifiers. The total number of cycles that were attributed to each layer is also shown in Figure 7. As expected, the production system layer accounted for most of the cycles. It is interesting to note that the classifier discovery layer, while not invoked at every cycle, was still responsible for the next largest block of cycles and that it grew with the number of classifiers. Consequently, the increase in total execution time was due to the classifier discovery layer.

Arithmetic operations accounted for most of the execution cycles. Communication operations accounted for very few.

In all simulations, primitive operations accounted for approximately 78 percent of the total number of cycles. In particular, it is important to know the degree of parallelism exercised within each of the primitives. Table 4 shows the ten primitives that consumed the most cycles, sorted in descending order by the percent of the total number of execution cycles they contributed. Next to each primitive is shown the average number of cycles executed per call; the percent of the total number of cycles; and, for a range of different numbers of classifiers, the average number of active PEs per call. From this table, it can be concluded that the arithmetic operations were directly responsible for the largest portion of the total number of execution cycles. Furthermore, it is also clear that the communication operations comprised an insignificant portion of the total number of cycles. The data for the random primitive is left out as that primitive was coded to generate a random number in all classifier records.

An important concern was whether there were enough active PEs to justify the use of bit-serial algorithms, or whether the work should have been performed sequentially in the controller with bit-parallel hardware. Those primitives where the average number of active PEs was less than the average number of cycles are *multiply-vector*, *scan add*, and *add-vector*. By moving these operations to the controller, approximately 12 percent of all cycles were eliminated. However, since the number of active PEs often varied greatly between individual calls, it was important to preface each routine with a test of the number of active PEs to determine where to perform the calculation.

Although this article focused on a single type of encoding for the classifiers, the architecture, while highly specialized, is quite capable of easily supporting any number of genetic algorithm encodings. This is due to the very flexible way in which the CAM data can be processed. Furthermore, the architecture will enable the development of many different algorithms for both the

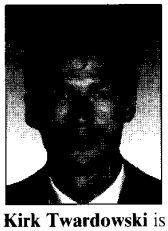
credit assignment and classifier discovery layers, in conjunction with new research results on LCSs.

The work reported here shows that associative architectures with the correct communication support, such as a reconfigurable long-distance communication bus, are effective for building Learning Classifier Systems. In particular, the experimental data showed that the runtime of the system increased only slightly even as the number of classifiers was increased sixfold.

Research to date has investigated the development of a specialized associative architecture to support inductive rule-based machine learning with genetic algorithms. Future development of intelligent systems with broad-based machine learning and adaptive capabilities may benefit directly from such specialized architectures. These architectures offer valuable potential for achieving a high degree of reactivity to inputs from the environment. In particular, as is possible with this architecture, it is important that ever-larger knowledge bases be supported in a manner that does not significantly affect runtime. ■

References

1. J.H. Holland, "Escaping Brittleness: The Possibilities of General-Purpose Learning Algorithms Applied to Parallel Rule-Based Systems," in R.S. Michalski, J.G. Carbonell, and T.M. Mitchell, eds., *Machine Learning: An Artificial Intelligence Approach*, Morgan Kaufmann, Los Altos, Calif., 2nd edition, 1986, pp. 593-623.
2. L.B. Booker, D.E. Goldberg, and J.H. Holland, "Classifier Systems and Genetic Algorithms," *Artificial Intelligence*, Vol. 40, Sept. 1989, pp. 235-282.
3. G. Robertson, "Parallel Implementation of Genetic Algorithms in a Classifier System," L. Davis, ed., *Genetic Algorithms and Simulated Annealing*, Pitman, London, 1987, pp. 129-140.
4. M. Dorigo, E. Sirtori, "Alecsys: A Parallel Laboratory for Learning Classifier Systems," *Proc. 4th Int'l Conf. on Genetic Algorithms*, Morgan Kaufmann, Los Altos, Calif., 1991, pp. 296-302.
5. C.D. Stormon et al., "A General-Purpose CMOS Associative Processor IC and System," *IEEE Micro*, Vol. 12, No. 6, Dec. 1992, pp. 68-78.
6. *Associative Computing: A Programming Paradigm for Massively Parallel Computers*, J.L. Potter, ed., Plenum Press, New York, 1992.
7. C.C. Weems et al., "The Image Understanding Architecture," *Int'l. J. Computer Vision*, Vol. 2, No. 3, Jan. 1989, pp. 251-282.
8. R.M. Lea, "WASP: A WSI Associative String Processor," *J. VLSI Signal Processing*, Vol. 2, No. 4, May 1991, pp. 271-285.
9. F.P. Herrmann and C.G. Sodini, "A Dynamic Associative Processor for Machine Vision Applications," *IEEE Micro*, Vol. 12, No. 3, June 1992, pp. 31-41.
10. R.H. Storer et al., "An Associative Processing Module for a Heterogeneous Vision Architecture," *IEEE Micro*, Vol. 12, No. 3, June 1992, pp. 42-55.
11. G.E. Blelloch, *Vector Models for Data-Parallel Computing*, MIT Press, Cambridge, Mass., 1990.
12. H. Muhlenbein, M. Gorges-Schleuter, and O. Krämer, "New Solutions to the Mapping Problem of Parallel Systems — the Evolution Approach," *Parallel Computing*, Vol. 4, No. 3, June 1987, pp. 269-279.



Kirk Twardowski is a staff engineer at Loral Federal Systems, Owego, New York. His research interests include high-performance computer architectures, associative processing, VLSI design, genetic algorithms, and artificial intelligence. He received a BS degree in computer systems engineering in 1986 from Rensselaer Polytechnic Institute, and MS and PhD degrees in computer engineering from Syracuse University in 1990 and 1994, respectively. He is a member of IEEE Computer Society, ACM, and AAAI.

Readers can contact the author at Loral Federal Systems, 1801 State Rt. 17C, Owego, NY, 13827, e-mail kirkt@lfs.loral.com.