Logic machines: A survey

by G.Z. QADAH
Northwestern University
Evanston, Illinois
and
M. NUSSBAUM
Institut fur Integrierte Systeme
ETH-Zurich/Switzerland

ABSTRACT

Logic(-based) programming languages are today the center of very many research efforts. One of these languages, PROLOG (PROgramming in LOGic), is used to program expert systems, natural language processors, computer aided design systems and compilers. A parallel variant of PROLOG is suggested as the language of the Japanese Fifth Generation computer project. Parallel to these efforts, a class of computer architectures that is suitable for supporting logic programming is emerging. Such a class is referred to as logic machines. In this paper, we propose a new taxonomy for the architectural space of logic machines. Based on such taxonomy, some of the proposed logic machine architectures are presented and compared.
INTRODUCTION

Logic Programming\(^1\,\,2\,\,3\) (or programming using logic formulas) is today the center of very many research efforts around the world. Two important features of logic make logic programming attractive, namely, the fact that logic is declarative\(^3\) and that it rests on a very powerful mathematical formalism. A language being declarative implies that the programmer needs only to specify what computations need to be performed rather than how they should be performed (the sequence of steps needed to carry them out). The language processor then decides, independent of the programmer, how such computations are to be performed. Declarative languages have many advantages, namely, higher programmer productivity and possibly high execution speed since novel hardware architectures can be used to support their execution.

The fact that logic rests on a very powerful mathematical base implies that many formal languages based on logic can be easily developed. In fact, in the last several years we have observed a good increase in the number of logic-based languages available to the programming community. In addition to PROLOG and its sequential and parallel variants\(^4\,\,5\,\,6\) concurrent logic-based languages such as PARLOG\(^2\) and Concurrent PROLOG\(^7\) have been developed. PROLOG and other logic languages have been used to construct expert systems, natural language processors, computer aided design systems,\(^8\) compilers,\(^10\) and event-driven simulators.\(^11\) A parallel variant of PROLOG, the Fifth Generation Kernel Language (FGKL),\(^4\) is suggested as the language of the Fifth Generation computer project.\(^12\)

The traditional implementation of logic programming systems as complex software systems running on general-purpose von Neumann computers, has resulted in slow and inefficient systems. One major reason for this is the fact that, in such implementation, the underlying hardware is general-purpose and sequential and not tuned properly to the requirements of such systems. The recent advances in VLSI technology, the dramatic drop in hardware prices, and the fact that logic programming systems lend themselves well to novel hardware architectures has inspired a new implementation. In such implementations, the general-purpose von Neumann computer is replaced with a dedicated machine tailored for non-numerical processing and, in most cases, utilizing parallel processing to support the logic programming systems. The aggregate of software and hardware components dedicated to the support of logic programming is referred to as a logic machine. Logic machines claim to improve the performance of logic processing through hardware specialization, increased parallelism, and increased processing power.

Recently, many architectures for the logic machine have been proposed. In this paper, we propose a new taxonomy for the architectural space of logic machines. Based on such taxonomy some of the proposed logic machines are presented and compared. The following section overviews the computational model as well as the inherent parallelism of logic programming. Next, a definition and a classification scheme (taxonomy) for logic machines is presented. An overview of some of the proposed logic machine architectures follows, and finally, we offer some general comments and concluding remarks.

THE COMPUTATIONAL MODEL OF LOGIC PROGRAMMING

Logic programming is a mathematical formalism based on horn clause logic suitable for expressing certain classes of problems requiring deductive reasoning.\(^1\,\,2\,\,3\) In the following, we present the elements of the computational model that underlie such programming environments as well as the various types of parallelism that exist in such a model.

Elements of the Computational Model

Conceptually, the computational model of logic programming consists of three elements, namely, a set of horn clauses, a set of goals (queries) and an inference (deduction) process. These elements are presented next.

The Horn Clauses

From a syntactical point of view and using the notations defined within the context of the logic-based language PROLOG,\(^7\) a (horn) clause has the general form

\[ S_0 := S_1, S_2, \ldots, S_n \]  

where \( S_i \) (\( i = 0, 1, \ldots, n \)) is a positive literal, "\( :-\)" is the implication operator, and "\( ,\)" is the logical AND operator. \( S_0 \) is the head literal (conclusion/goal) of the clause, while \( S_i \) (\( i = 1, 2, \ldots, n \)) denotes the body literals (subgoals) of the clause. The literal in a clause is an expression of the general form

\[ p(t_1, t_2, \ldots, t_m) \]  

where \( p \) is a predicate (relation) or functional symbol and \( t_i \) (\( i = 1, 2, \ldots, m \)) is a term. A term is either a constant whose symbol starts with a lower case character, a variable whose symbol starts with an upper case character, or an expression of the same form as (2) except that \( p \) can only be a functional
symbol. The variables in horn clauses are typeless. That is, they may assume different types throughout the process of manipulating the clauses within which these variables are defined.

From a semantic point of view, a literal may have only one of the two logical values, true and false, and the horn clause of (1), therefore, is interpreted as:

\[ S_0 \text{ is true if } S_1 \text{ is true AND } S_2 \text{ is true ... AND } S_n \text{ is true.} \]

However, when a clause contains zero body literals, it is interpreted as “\( S_0 \text{–true.} \)” That is, “\( S_0 \text{ is (always) true.} \)” Such a clause is called the unit clause, assertion or tautology.

Horn clauses can express both simple and complex knowledge about objects in the real world. The unit clause expresses a simple (atomic) fact about an object. For example, the unit clause,

\[ \text{like(arthur, john):} \]

expresses the fact that “arthur likes john.” Such a clause is called the ground unit clause. Replacing “john” of clause (3) by a variable \( X \), the new clause, the non-ground unit clause, expresses the fact that “arthur likes \( X \) irrespective of the value that \( X \) might have.” That is, “arthur likes everyone and everything.”

The more complex facts about objects are expressed using clauses of the general form presented in (1). For example, the clause

\[ \text{uncle(bob, ruth):} \text{• sister(ann, bob), mother(ann, ruth)} \]

expresses the fact that “bob is the uncle of ruth” if “ann is the sister of bob” AND “ann is the mother of ruth.” These types of facts are implicit in the sense that the body of the clause must be tested and proved to be true in order to conclude that the head literal is true. A more interesting case arises when the constants inside the literals of clause (4) are replaced by variables, as follows:

\[ \text{uncle}(X, Y):\text{• sister(Z, X), mother(Z, Y).} \]

Such a clause then expresses a general “rule” that applies to members of general classes. In the above case, the rule partially defines the “uncleship” relation between human beings (a class of objects) in terms of the simpler relations, “brothership” and “fathership,” defined on the same class.

Figure 1 shows a set of horn clauses. Any literal in such a set is of the form \( r\text{-symb}(X, Y) \) and can be read as “\( X \) is \( r\text{-symb} \) of \( Y \).” Most examples and illustrations in this rest of this paper will be based on this set.

**The Goal/(query)**

The goal is a logical statement whose truth value needs to be determined with respect to a set of horn clauses. The statement is true if it is a logical consequence of such set, otherwise, it is false. A goal has the following general format:

\[ :-S_1, S_2, \ldots, S_n \]

That is, it is a horn clause with zero head literal and one or more body literals. When a goal contains no variables (a ground goal), its answer is simply true (if a proof can be found), or false (otherwise). For example, invoking the set of clauses of Figure 1 with the following goal,

\[ :-\text{uncle(bob, ruth)} \]

yields the answer “true,” since “bob is uncle of ruth” is a logical consequence of the clauses \( C_3, C_7 \), and \( C_{10} \) of Figure 1. A more general situation occurs when the goal contains one or more variables, as in the following:

\[ :-\text{uncle}(X, \text{ruth)} \]

In such a case, the answer to the goal is a set of patterns of values to the variables in the goal under which such goal is true, if any, otherwise the answer is false. Actually, the requirement of returning patterns of values to the variables in the goal (if any variable exists) rather than returning only true or false is one of the major differences between the computational models of theorem proving and logic programming. 14

When the goal statement contains variables, it is more appropriate for the goal to be called a query since such a statement can be viewed as a specification of the set of value patterns under which the statement is true. The value patterns for the variables in a query are called solutions. Invoking the set of clauses of Figure 1 with the query of (5), yields the answer “true” and the solution \( X = \text{bob} \).

**The Inference Process**

The inference (deduction) process takes a goal and a set of horn clauses and tries to prove (infer) that such a goal is true with respect to such set. Such a proof involves the establishment of the fact that the input clause (query) is consistent with (or a logical consequence of) the set of horn clauses. One of the most common inference methods that is suitable for horn clause logic is the one based on the resolution principle. 15
At the heart of this method is the resolution (reduction) step. Such a step can be decomposed into two steps, unification and substitution. Unification is the process of making two literals identical by replacing their free variables with a common set of binding values, called the unifier. The process may succeed or fail; however, if it succeeds, then it generates the unifier. For example, the two literals, “brother(fred,larry)” and “brother(Z,larry)” are unifiable under the binding set \{Z/fred\} to yield the common literal “brother(fred,larry).” The two literals “brother(Z,larry)” and “brother(Z,Y)” are also unifiable under the set of bindings \{X/Z, Y/larry\}. On the other hand, the two literals “brother(Z,larry)” and “brother(X,joe)” are not unifiable because there exists no set of bindings for the free variables of the two literals which make them identical (the second term in each of the literals has a different constant value). For the same reason, the two literals “brother(fred,larry)” and “father(fred,larry)” are not unifiable (the predicate symbol is not the same in both literals). An important feature of the unification process is that it permits a bidirectional binding of a variable from one literal to a constant, a variable, and even to a general term of the other participating literal.

It is also important to notice that having the same predicate symbols and an equal number of terms is a necessary (but not sufficient) condition for two literals to unify. A general algorithm for the unification process can be found in Sterling and Shapiro. 16

The unification of a literal from a goal with a general clause is performed by unifying the goal literal with the head literal of the clause and generating the binding set. For example, unifying “uncle(X,ruth)” from the goal “:-uncle(X,ruth)” with

\[
\text{uncle}(X,Y) :- \text{brother}(Z,X), \text{father}(Z,Y) \tag{6}
\]

will succeed and generate the binding set \{X/X, Y/ruth\}.

The substitution step is to replace the literal in a goal with the body literals of a clause that has unified with it; then each variable in the new expression is replaced with its value from the binding set (unifier). The expression resulting from the substitution step is a new goal and is called the “resolvent.” If no body literal exists (the clause is a unit clause), then, the substitution step replaces the literal of the goal with the logical value “true” (the interpretation of the empty body of a unit clause). Reaching an empty clause indicates that the goal is indeed derivable from the set of clauses (subject to whatever bindings are made to the variables in the goal “:-father(X,ted)” of the goal “:-father(X,ted)” unifies with the clause “father(paul,ted):-.” The substitution step results in a resolvent of the form “:-true.” Such a resolvent is called the empty (or null) resolvent (a clause with no head or body literals) and is given the symbol “[].”

The resolution process can be best described through the algorithm shown in Figure 2. The proof of a goal G, using such a process, starts by selecting one of the goal’s literals g (step 1) and finds a clause c from the set of clauses that unifies with g and the corresponding unifier \theta (steps 2 and 3). Applying, then, the substitution step to the goal G and the clause c using the unifier \theta results in a resolvent R (step 4). Each time a new resolvent is derived (or in other words, a goal is reduced), the former steps are repeated, with the new resolvent as its input goal, until eventually one of the two states is reached, namely, the new resolvent is an empty or a non-resolvable one. (A resolvent is non-resolvable if it cannot unify with any clause in the set of logic clauses.) Reaching an empty clause indicates that the goal G is indeed derivable from the set of clauses (subject to whatever bindings are made to the variables in the

---

From the collection of the Computer History Museum (www.computerhistory.org)
goal statement). The sequence of resolution steps that terminates with "[]" is called a "success" proof sequence and the set of bindings that has been made to the variables of the goal throughout such sequence represents a solution to the input goal (query). On the other hand, reaching a non-resolvable clause indicates the failure of the sequence to prove the goal G. The sequence that terminates with such a clause is called a "failure" proof sequence. If neither the empty nor the non-resolvable clause is reached, the inference algorithm loops forever, producing an infinite proof sequence. Figure 3 shows a sequence of resolution steps generated by the inference process in response to the goal ":-uncle(X, ruth)," using the algorithm of Figure 2. Such proof sequence starts by unifying "uncle(X, ruth)," with C_{10} of Figure 1 to yield the resolvent ":-sister(Z, X), mother(Z, ruth)," then unifying "mother(Z, ruth)" with C_7 to yield the resolvent ":-sister(ann, X)" and finally unifying "sister(ann, X)" with C_5 to yield the empty resolvent. That is, the statement "uncle(X, ruth)" when "X = bob" is indeed a logical consequence of the set of clauses presented in Figure 1.

The set of all proof sequences (success, failure, and infinite) for a goal with respect to a set of clauses forms a space, the search space. Such a space can be represented as a tree, the search tree. Figure 4 shows such a tree for the goal ":-uncle(X, ruth)." The root of the tree is the goal to be proved, while the rest of the nodes represent resolvents. A leaf node (if it exists) represents an empty resolvent of a non-resolvable resolvent. The arc between a node and one of its children corresponds to a resolution step and is labeled by the set of bindings generated during such step. The nodes at the i-th level of the tree represent the set of all resolvents that can be obtained from the goal in i resolution steps. The path from the root of the tree to a leaf represents the sequence of resolution steps that leads to that leaf together with the sets of bindings generated throughout such sequence. A tree may contain sequences that terminate with success (1 → 2 → 3 for example), with no success (1 → 4 for example) or sequences that never terminate (not present in Figure 4). An important feature of the search tree is that it is an "OR" tree. That is, to prove that any non-leaf node (resolvent or goal), in the tree is true, it is sufficient to prove that any of its children nodes (child, "OR" child1, "OR" child2, "OR"...) is true.

The resolution process is non-deterministic since at any step of the proof, the selection of a literal from those of a resolvent (step 1 of Figure 2) as well as the selection of a clause out of the set of ones unifiable with g (step 3) to participate in a resolution step, is performed in a non-deterministic fashion. That is, the resolution process makes, at these steps, the correct choice of a literal and a clause that leads to a solution. Therefore, the non-deterministic resolution process can be viewed as a process that finds (through making, somehow, the correct choices at the non-deterministic points) the successful proof sequences from all of the other ones in the search tree. The resolution process is semantically very powerful, since it guarantees finding all possible sequences leading to empty clauses (solutions) even in the presence of infinite sequences. Such power is not without a price. A non-deterministic process cannot be implemented (however, it can be simulated or approximated). In addition, both space and time complexities of such processes are exponential in terms of number of levels in the search tree and, therefore, invoking even a small set of clauses with a goal can be very involved computationally. Practical logic-based systems try to implement deterministic algorithms that are equivalent in semantical power to (or even weaker than)* that of the presented algorithm but have improved space and/or time requirements. These systems simulate or approximate the non-determinism in the resolution process using procedures that search the different paths of the search tree (or a more efficient representation of the search space) for solutions. Such systems are presented next.

**Parallellism in Logic Programming**

The resolution process of logic programming contains many activities with ample embedded parallelism. Step two of the algorithm presented in Figure 2 finds a set of qualified clauses

*PROLOG, for example, implements an inference algorithm that, under certain conditions, fails to generate solutions in the presence of infinite sequences.
S, each of which can unify with the literal of the current goal. Instead of selecting one of these qualified clauses to reduce the goal (producing a new resolvent), a procedure simulating the inference process can proceed to reduce the goal with two or more of the qualified clauses in parallel. Such simultaneous activities are referred to as *or* parallelism. Using such parallelism, for example, results in the simultaneous reduction of the goal ‘‘:-uncle(X,ruth)” using the two rules, ‘‘uncle(X,Y):-brother(Z,X),father(Z,Y)” and ‘‘uncle(X,Y):-sister(Z,X),mother(Z,Y)” of Figure 1. The *or* parallelism, when used, permits the inference process to find different solutions for the same goal (query), in parallel. The *or* parallelism is simple to exploit since *or* parallel activities, once initialized, do not interact with each other. One problem with the *or* parallelism is that when it is utilized recursively at each goal in the search tree, the generated parallel activities grow exponentially with respect to the number of levels in the search tree. The exponentially-generated activities are beyond the capability of any practical parallel processing system, and therefore, some methods have to be developed to restrict such parallelism.

A second type of parallelism is the so called *and* parallelism. Such parallelism corresponds to the simultaneous solution of two or more literals (subgoals) in a given goal (resolvent). Using such parallelism, for example, a solution for each of the subgoals ‘‘:-brother(Z,X)” and ‘‘:-father(Z,ruth),” of the goal is found in parallel. That is, the two search trees which correspond to the former subgoals are constructed and searched simultaneously to find a solution for each of the subgoals. When the literals in a goal have no shared (common) variables, then the solution for the goal is simply the concatenation of the individual solutions obtained for each of the literals in the goal. However, when shared variables between the literals exist (variable Z is shared between the two literals of the goal ‘‘:-brother(Z,X),father(Z,ruth),” special care must be taken. A solution for the goal is not obtained by simply concatenating the individual solutions but by obtaining from them a solution in which the bindings for the shared variables are the same. To illustrate this point, consider obtaining the solution {Z|ted, X|bob} for the subgoal ‘‘:-brother(Z,X),” and the solution {Z|paul} for the subgoal ‘‘:-father(Z,ruth).” A solution for the conjunction of the two subgoals does not exist, because the two solutions bind different values for the shared variable Z. On the other hand, the solution {Z|ted} for the subgoal ‘‘:-father(Z,ruth)” together with the previous solution for the other subgoal produces a solution for the goal because the shared variable Z has the same value in both of the individual solutions. The problem of shared variables complicates, to a large extent, the utilization of *and* parallelism by a parallel processing system.

A third type of parallelism is the so called *search* parallelism. This parallelism corresponds to the simultaneous search of the set of clauses for those that can unify with a given literal (SIMD search parallelism and can be used to initialize *or* parallel activities) or the simultaneous search for clauses that can unify with different literals (MIMD search-parallelism and can be used by the *or* parallel or *and* parallel activities). *Search* parallelism is very important for parallel logic systems—especially those that contain very large logic bases.

A fourth type of parallelism is the so called *unification* parallelism. This parallelism corresponds to the parallel activities within the unification algorithm. In general, the amount of this type of parallelism is very small since the unification operation tends to have a rather sequential nature. However, the *unification* parallelism can be of some advantage when both of the literals that participate in the unification operation contain many terms, each of which has a relatively complicated structure.

**LOGIC MACHINES: GENERAL ARCHITECTURE AND A TAXONOMY**

The entity ‘‘logic machine” can be defined as an aggregate of software and hardware components designed to simulate or approximate the computational model of logic programming. By the word ‘‘simulate” we refer to those systems that attain the full semantical power of logic programming (that is, for a query and a set of clauses, these systems generate all the solutions that can be generated by the non-deterministic inference process). By the word ‘‘approximate” we refer to those systems that implement a model of computation close to that of logic programming, but have less semantical power than logic programming (that is, they may not generate all possible solutions). In the following, an abstract (general) architecture and a taxonomy for logic machines proposed so far are presented.

**General Architecture**

From an architectural point of view, a logic machine, Figure 5 is organized into two major components, namely, the logic programming system and the underlying hardware system. In addition to the (user) queries (goals), the workload of such a computer system includes operations to manage, update, and maintain the knowledge stored in such a system. The logic programming system contains components for implementing or approximating the computational model of logic programming. These components, as shown in Figure 5, are the logic base (program) and the inference procedure.

**The Logic Base**

In general, the logic base (program) consists of a set of logical statements which express certain facts about a collection of real world objects and the relationships that exist between these objects. These statements are taken from a language, a logic programming language, which serves as a tool for the user (programmer) to specify a logic program. Basically, these statements are horn clauses extended to include some extra information which helps the inference procedure (see the following section) to perform a more efficient search of the search space and/or to provide some explicit information about the sequence in which the literals in the body of a clause or the set of clauses in the logic program are to be processed by the inference procedure. For example, in the logic programming language PROLOG the CUT operator ‘‘!:” helps PROLOG’s inference procedure to trim the space
that is being searched for solutions for a given query. In PARLOG, a parallel logic programming language, the literals in a clause are separated by either the operator "&" or the operator ",". The former indicates to the inference procedure that the literals within the body of a clause separated by such operator are to be executed sequentially, whereas the literals separated by the operator "&" are to be executed in parallel. In a similar fashion, the horn clauses in PARLOG are separated by the operator ";" or ";". Two clauses separated by ";" are to be processed serially, whereas those separated by ";" are to be processed in parallel.

The Inference Procedure

The inference procedure simulates or approximates the resolution process of logic programming. For a given query, this procedure searches a more efficient representation of the search space than that of the search tree. One source of inefficiency in the resolution tree is the fact that it contains proof sequences which generate the same solutions (redundant proof sequences) for the same query and logic base. For example, in Figure 4, the two sequences 1 → 2 → 3 and 1 → 5 → 6 lead to the same solution for the query ":-uncle(X, ruth)," namely, "X = bob." An efficient inference procedure needs to generate (search) only one of these sequences but not both. The tree which has no redundant proof sequences is called the proof tree (another OR-tree). Such a tree is obtained by expanding only one literal from each resolvent at each level of the search tree (rather than by expanding all of the literals to generate all possible resolvents). Depending on which literal from each resolvent is expanded, a number of different proof trees can be obtained. These trees are equivalent (that is, if a solution for the query can be obtained from one of these trees, then, the same solution can be obtained from every other tree) but have a different total number of proof sequences. A smart inference engine can take advantage of such arbitrary choice to generate and search the proof tree which has the minimum number of proof sequences. Figure 6 presents two possible proof trees for the goal ":-uncle(X, ruth)." These trees are generated by reducing the underlined literals in Figure 6 first. It is easy to see that searching both trees will generate the same solutions, however, searching the tree of Figure 6(b) will take much less time than searching the tree of Figure 6(a).

Still another source of inefficiency in the OR-tree representation of the search space is the high number of branches coming out from a node in such a tree (this factor is important for the efficient implementation of both sequential and parallel inference schemes. To illustrate this point, consider the node (resolvent) ":-brother(Z,X),father(Z,Y)" and assume that only facts can unify with each of these literals and the number of these facts are n and m, respectively. The number of branches which come out from such a node equals 0(n × m). To overcome such a high factor, a new, more efficient representation for the search space has been introduced, the AND/OR tree. The basic principle underlying such representation is the replacement of each non-leaf node in the OR tree, such as the one shown in Figure 7, by two levels of nodes. The first level has only one AND node and the second level has as many OR nodes as the number of literals in the resolvent. The AND node is labeled by the conjunctive resolvent itself. The name AND is given to a such node because in order to prove that such a node is true, we have to prove that all of its children nodes (child1 and child2 and . . .) are true. Each OR node represents a resolvent of one literal (unit resolvent or subgoal) and needs to have only one of its children nodes to be true in order for itself to be true. A node in such representation has only 0(n + m) branches, a substantial reduction over that of the OR-tree representation. Figure

Figure 6—Two possible proof trees for ":-uncle(X, ruth)"
The search for solutions in such a tree is much more complicated with respect to the logic base of Figure 1. The AND/OR tree representation is not without problems. The search for solutions in such a tree is much more complicated than that of the OR-tree representation, since a solution in this representation has the form of a subtree rather than a sequence of branches from the root to a null leaf node. For example, the subtree $1 \rightarrow 4 \rightarrow 5 \rightarrow 4 \rightarrow 2 \rightarrow 3$ of Figure 8 represents the solution for the query "$\neg \text{uncle}(X, \text{ruth})$". To search the OR-tree for solutions one needs to go only top down, but to search the AND/OR tree for solutions one needs to go first top-down until the leaf nodes are reached, then the search continues bottom-up. Despite the complex search problem, most of the inference procedures in logic programming systems use the AND/OR tree representation of the search space because of its low branching property.

Many logic machines have been proposed so far. These machines can be viewed as points in an architectural space, the logic machine space. This space, Figure 9, is defined by two attributes which characterize the abstract architecture of Figure 5, namely, the search strategy and the hardware organization. The attribute search strategy specifies partially or completely the method by which the inference procedure in a logic machine performs the search of the AND/OR or the OR tree, respectively, for finding solutions. On the other hand, the attribute hardware organization specifies the way the hardware of the logic machine is organized to support the search of the tree. Three possible methods exist for searching a tree, namely, depth-first (DF), breadth-first (BR), and dynamic (DYN). In the depth-first method and starting from the root of the tree, the most recently generated children of an or node (or goal) get searched (reduced) first. In the breadth-first search, the children of an or node get reduced in the order in which they are generated. In the dynamic search, the children of an or node are reduced in an order determined by some criteria. Such criteria can be preprogrammed in the inference procedure or specified by the user through the statements of the logic base (program). For the OR-tree representation of the search space, the previous methods fully specify the search strategy of the inference procedure since such tree contains nodes of only the or type. However, this is not the case for the AND/OR tree and in order to completely specify the search strategy of such a tree, one must specify the method by which the children of an and-node (subgoals) get reduced. We have omitted here such specification to keep our classification scheme as general as possible. However, such specification will be delayed until we present some of the proposed logic machines.

The hardware of a logic machine can be organized in three different ways, namely, as a single instruction stream-single data stream (SISD), single instruction stream-multiple data stream (SIMD), or multiple instruction stream-multiple data stream (MIMD) machine. A logic machine in the SIMD class is simply a classical von Neumann processor programmed to perform the serial search of the tree. Because of its serial nature, such a processor can be active at only one node of the tree at any point in time. The processor can be a classical processor with general-purpose instruction set or a special purpose processor that is tuned for efficiently implementing the serial search of the tree. Such tuning varies from simply extending the processor instruction set to include some more suitable instructions for the symbolic processing environment, all the way up to designing such a processor around a radically different instruction set which is more appropriate for supporting the logic processing environment. In addition, such a processor may include specialized hardware that takes advantage of the small amount of parallelism which exists in the processing of a node in the tree, such as unification parallelism, pipelined instruction, execution, and unification coprocessing.

A logic machine in the SIMD class is organized as an array of simple processors, each with its own local memory. These processors are controlled and managed by a single master processor. At any point in time, all the array processors are
performing the same task on the different data elements stored in the processors' local memories. The master processor can have a general or special instruction set. An SIMD machine is a good search engine since the search operation can be performed in parallel on all the data elements in the array processors. Storing the logic base in the local memories of the array processors permits the efficient and parallel implementation of many search-based operations such as the unification operation. The SIMD class of logic machines is not popular since it can take advantage of only one type of logic programming parallelism, namely, the search parallelism (especially when processing very large logic bases). Such organization cannot take advantage of the other types of parallelism that exist in logic programming.

A logic machine in the MIMD class is organized as a set of independent processors intercommunicating over an interconnection network. The processors can be general-purpose or special-purpose ones and different types of interconnection networks\textsuperscript{26} may be used. An important characteristic of this class of machines is the ability to perform one or more tasks in parallel. Actually, through the special design of the processor itself and through the interconnection of many of these processors together, such a machine can use not only the micro-parallelism but also the macro-parallelism imbedded in logic programs to speed up its evaluation. One important aspect of an MIMD machine is the scheme by which the machine's parallel activities are controlled and synchronized. Two broad classes of schemes have been developed, each is based on one of the basic concepts; control-flow and data-flow.\textsuperscript{27,28} The former scheme is the parallelized version of the traditional control scheme adopted for the classical von Neumann processor. A computation in such a scheme is controlled directly by the programmer through the programming language. On the other hand, a computation in a data-flow based scheme is performed only when all of the computation's input data is available, thus permitting as much parallel activity as possible. The advantages and disadvantages of both of these approaches can be found in Arvind and Iannucci,\textsuperscript{29} and in Gajski, Padua, Kuck, and Kuhn.\textsuperscript{30} Both of these control schemes have been used by designers to develop MIMD logic machines.

Figure 10 presents some of the proposed logic computers classified according to the above scheme. Guided by such a classification scheme, an overview of a sample of the architectures will be presented next.

OVERVIEW OF SOME LOGIC MACHINES

The Depth-First-SISD Logic Machines

The logic machines in this class search the AND/OR tree representation of the search space using a single processor and the depth-first strategy to search the children of both the or nodes and and nodes of such representation. In processing the children (subgoals) of an and node using such a strategy, one can use one of two techniques: one-solution-at-a-time and all-solutions-at-a-time. Using the first technique, as shown in Figure 11, a solution is first obtained for subgoal. The solution is then used to instantiate the variables that are common to both the first and the rest of the node's subgoals. The processing of subgoal\textsubscript{1} is frozen and the processing of the second subgoal for finding a solution is activated. This process is repeated for all the subgoals of the and node. A solution to the and node is obtained if a solution is obtained for every subgoal in that node. Obtaining a new solution for any subgoal of the and node, subgoal, for example, is carried out using a technique called backtracking. Using such a technique, the subgoal is first tried for a new solution using the old instantiations. If this fails, then the activation backtracks to the previous subgoal to obtain a new solution for that subgoal. Such a solution is then used to instantiate the variables in subgoal, and a new solution is tried. Such backtracking may propagate recursively to one or more of the subgoals previous to subgoal.

Processing an and node depth-first using the all-solution-at-a-time technique is performed, as shown in Figure 11, through first obtaining all the solutions for subgoal. This set is then used to instantiate the variables that are common to the first and the rest of the node's subgoals. Subgoal\textsubscript{1} is then eliminated and the control is transferred to subgoal\textsubscript{2} to find all of its solutions. Such a process continues until all the subgoals of the and node have been processed. The set of solutions which has satisfied all the subgoals is then the solution set for the corresponding and node. It is important to note that using
the all-solutions-at-a-time approach, the backtracking is no
longer needed for processing the subgoals of an and node.

The search of the AND/OR tree representation using the
depth-first strategy at both the and and or node levels and the
use of the backtracking mechanism form the kernel of the
inference procedure of PROLOG,13 today's most known logic
programming language (system). Such search strategies have
permitted an elegant stack-based implementation for such
inference mechanisms with excellent memory management
schemes.22 This implementation may take one of two forms:
interpretation based or compilation based. In the former, the
inference procedure is implemented as an interpreter pro­
gram (written in some high-level language) and runs on a
single general-purpose processor (the host). Such a program
interprets the user query using the logic base to produce solu­
tions. On the other hand, the compilation-based system in­
cludes a compiler which compiles the user query and the logic
to produce an object program which runs on the gen­
eral-purpose processor.31 The early implementations of
PROLOG followed one of these methods. Although the
compiler-based implementation is much faster than that of the
interpreter-based implementation, nevertheless, both of these
implementations resulted in slow PROLOG systems. To im­
prove performance, several recent implementations which
augment or replace the general-purpose processor with more
suitable hardware have been proposed. PSt5,32 and PEK33 are
special purpose PROLOG processors which implement the
PROLOG Interpreter as a microprogram. In Robinson,32 and
Woo,36,37 the general-purpose processor is augmented with a
special-purpose hardware unification unit. Since the uni­
fication is a frequent operation when executing PROLOG pro­
systems, speeding up the execution of such operations results in
a faster overall system.

PLM38,39 and HPM40 are true special-purpose PROLOG
processors. These processors have been built to execute an
instruction set, proposed by Warren,4 specially designed to
support PROLOG, its depth-first search strategy, and its
backtracking mechanism. The instruction set is at a higher
level than ordinary general-purpose instruction sets and in­
cludes instructions which directly perform the unification,
memory management, and so forth. The execution of a query
on such a processor is carried out by first compiling the query
together with the logic base into statements using Warren’s
instruction set. Then the resulting program is executed by the
specially designed processor. To further improve the execu­
tion speed of PROLOG programs, Tick and Warren42 have
proposed a pipelined PROLOG processor. Such a processor is
essentially the same as PLM or HPM except that it is de­
signed to pipeline the execution of Warren instructions.

The Breadth-First-MIMD Logic Machines

The logic machines in this class unfold and search the or
nodes of the OR or the AND/OR tree representation of the
query’s search space using the breadth-first strategy. As a new
level of or nodes is unfolded, one or more processes are as­
signed to search such nodes, thus creating a set of parallel
processes which cooperate to produce solutions. Such an as­
sembly of processes is assigned to the different processors of
a multiprocessor machine for execution. The parallel OR and
parallel AND/OR machines43,44 are good representatives of
this class and will be presented next.

The parallel OR machine43,45 searches the OR-tree
representation for solutions. The process assigned to an or
node (the root node in Figure 6(a), for example) generates all
of its children nodes by performing a resolution step on one of
the node’s literals and assigns a process to each one of the
newly generated nodes (resolvents). The children processes
inherit the binding environment from the parent process;
thereafter the parent process is eliminated and each of the
children processes repeats the parent’s action. Such activities
continue until a leaf node is reached by a process; thereafter,
a new process for the leaf node is not generated, but rather a
solution is reported to the system if the leaf node is of the null
type. The assembly of processes execute on a multiprocessor
system. Such execution is controlled through an elaborate
token-based scheme.45

The parallel AND/OR machine44,20 searches the AND/OR-
tree representation for solutions. A process, the and-process,
assigned to supervise the execution of an and node (the root
node in Figure 8, for example), generates one or-process for
each one of its children nodes (or, in other words, for each of
the literals in the body of the corresponding resolvents). An
or node (process), in turn, generates an and-process to super­
vise the execution of each of its children and nodes (conjunc­
tive resolvents). For example, the or-process P5 associated
with the node “:-uncle(X, ruth)” in Figure 8, generates two
and-processes P4 and P3 to supervise the execution of each of
the resolvents “:-brother(Z, X), father(Z, ruth)” and
“:-sister(Z, X), mother(Z, ruth),” respectively. As a solution
is obtained for the variables in the literal of an or-process (the
literal “sister(Z, X)” of Figure 8, for example), a “success
message and the obtained solution are reported to the parent
and-process P5. The P5, then, invokes the or-process associ­
ated with its second child P3, passes to it the variable bindings
and requests a solution to the remaining free variables of the
associated literal. P3 is then blocked. When an and-process
receives success messages from all of its children, it passes on
a similar message to its parent together with the obtained
bindings for the free variables. The and-process is then
blocked. Such activities continue until the and-process of the
root receives a “success” message from all of the or-processes
associated with its children nodes. The bindings to the free
variables in the query form a solution for the query.

The parallel AND/OR scheme is designed to run on a
loosely coupled multiprocessor (no shared memory is re­
quired) in which processors (processes) exchange information
via message passing. Many improvements to this basic scheme
have been proposed. In Furukawa, Nitta, and Matsumoto,46
the parallel AND/OR scheme is modified such that when a
process reports a “success” message and a solution to its
parent process, it is ordered to continue finding a second
solution. The process is then blocked only when the second
solution is found and the parent node is still processing the
first one (limited form of pipelining).

In the parallel AND/OR scheme reported in Lindstorm and
Piangaden,47 a process never gets blocked, but rather it
continues supplying its parent process with solutions until it runs out of them; thereafter such a process is eliminated. Such a scheme includes many more parallel activities than the AND/OR model of Conery and approaches the amount of parallelism encountered in the parallel OR model of Ciepielewski.\textsuperscript{23}

DeGroot\textsuperscript{48} suggests the partitioning of the set of literals in the body of a resolvent (the children of an and node) into subsets such that no literals with shared variables can exist in two different subsets. The independent subsets are then processed in parallel. The new scheme is stack-based which eliminates the need to generate the large number of processes encountered in Conery’s scheme.\textsuperscript{20} It also eliminates the need to pass information around via messages; a large reduction in the overhead encountered when processing a query.

\textit{The Dynamic-MIMD Logic Machines}

The logic machines in this class unfold and search the nodes in the tree representation of the query search space for a single solution in an order determined by some criteria. Such criteria can be specified by the user through the statements of the logic base (program) as in the concurrent logic-based languages, Concurrent PROLOG\textsuperscript{8} and PARLOG\textsuperscript{7} or preprogrammed in the inference procedure as in Li and Wah\textsuperscript{20} and Lipovski and Hermenegildo.\textsuperscript{49} In the latter case, a heuristic function\textsuperscript{49} guides the search of the inference procedure for a solution.

Concurrent PROLOG\textsuperscript{8} and PARLOG\textsuperscript{7} bear a close resemblance to each other. They are both designed to execute on parallel machines. These logic systems use an extended version of horn clauses—the guarded horn clauses. Such a clause has the following format:

\[ S0 \leftarrow C1, C2, \ldots, S1, S2, \ldots \]

where \( S0 \) and \( S1, S2, \ldots \) are the same as in an ordinary horn clause. \( \leftarrow \) is the guard operator and \( C1, C2, \ldots \) is a conjunction of literals which form what is called the guard condition. Procedurally, the above clause states that \( S0 \) can be replaced with \( S1, S2, \ldots \) only after the guard condition \( C1, C2, \ldots \) evaluates to “true.” Or, in other words, control is passed from \( S0 \) to the conjunction of literals to the right of the guard (the body of an ordinary horn clause) only after the conjunction of literals to the left of the guard evaluates to “true.” The literals in the guard condition are not allowed to instantiate any variables as they execute.

The Concurrent PROLOG and PARLOG search the AND/OR tree representation. As a new level of and nodes in the tree is unfolded (refer to Figure 8), each node is assigned a process, an and-process, to supervise the execution of the body of the corresponding guarded clause. Every and-process, in parallel, evaluates the guard condition associated with the corresponding and node. The first process to evaluate its guard to “true” is allowed to unfold the tree further (that is, the control is transferred from \( S0 \) to \( S1, S2, \ldots \)), while all of the other competing processes are cancelled. An and-process evaluates a guard condition by assigning one or-process to supervise the execution of each of the literals in such a guard condition. In Concurrent PROLOG,\textsuperscript{8} the or-processes evaluating the different literals of a guard execute in parallel. In PARLOG,\textsuperscript{7} the programmer can specify the sequence (serial or parallel) in which the or-processes evaluate a given guard. The programmer can also specify the sequence in which the different and nodes of the same level execute. PARLOG has been implemented on a reduction-based multiprocessor system called ALICE.\textsuperscript{20} Recently, a new machine consisting of a set of processors, each specially designed to execute PARLOG, is being investigated. A processor in such a machine is built to support a newly proposed instruction set\textsuperscript{51,52} especially designed to support the process-based search strategy of PARLOG. A multiprocessor called the Bagel\textsuperscript{53} has been proposed to execute Concurrent PROLOG. The Bagel is a set of transputers\textsuperscript{54} interconnected through a modified mesh network called the torus.\textsuperscript{55,56}

The search strategy adopted by Concurrent PROLOG and PARLOG replaces the notation of nondeterminism (taking the right choice) of the resolution process with indeterminism (taking an arbitrary choice). That is, to find a solution for a query only one path in the tree is tried. If the path leads to a solution, then the query succeeds and this solution is returned to the user; otherwise, the query fails without trying to search the other paths in the tree. Because of adopting such a search strategy, both Concurrent PROLOG and PARLOG are semantically weaker than logic programming. That is, even if a solution exists for a query, there is no guarantee that these systems will be able to produce such solution. Finding such a solution depends largely on the way the various generated processes are scheduled for execution.

MANIP-2,\textsuperscript{53} a multicomputer system consisting of a set of computers interconnected through a global broadcast bus and a selection and redistribution network, has been proposed to process logic programs in parallel. It implements a parallel heuristic search of the AND/OR or the OR/AND tree representation of the logic program for finding a single solution. Such a search is guided by a heuristic function that uses the ratio of the success probability of a literal (or node) to the estimated overhead of evaluating such literal.

\textbf{COMMENTS AND CONCLUDING REMARKS}

In this paper we have presented a taxonomy for a class of computer architectures called logic machines that use special-purpose hardware and/or parallel processing to speed up the processing of logic. Guided by such a taxonomy, some of the proposed logic machines have been overviewed. Some general comments concerning the proposed logic machines are presented below.

The taxonomy presented in the section on parallelism groups the logic machines, proposed so far, according to the strategy of search carried out by the corresponding machine; the machines fall into three classes, namely, the depth-first, breadth-first, and dynamic. From a semantic point of view, machines that use breadth-first search implement the computational model of logic programming faithfully. How-
ever, those machines that use the depth-first search implement a model of computation weaker than that of logic programming. Unless special care is taken, when infinite branches are present in the search tree of a logic program, those machines cannot produce some or all of the solutions that can be generated by the non-deterministic resolution processes. Depending on the heuristics adopted by the logicle machine in the dynamic class, such a machine may implement faithfully the model of logic programming, as in MANIP-2,23 or much weaker computational models as those implemented by PARLOG7 and Concurrent PROLOG.8

Machines in the breadth-first class have exponential storage and execution time complexities with respect to the number of levels in the search tree of logic programs. Such complexities remain exponential even when a polynomial number of processors are used in the execution of logic programs.57 The exponential storage requirement is a severe drawback of machines in the breadth-first class, especially when the search tree is deep. Some mechanisms must be designed to limit the maximum width of the tree during execution as in Epilog58 logic system. Machines in the depth-first class require exponential time complexity and linear storage. This is a great plus to this type of machine. Machines in the dynamic class require variable time and storage complexities depending on the heuristic function being used in association with the particular machine.

From the parallelism point of view, the breadth-first machines can take advantage of all the parallelism embedded in the computational model of logic programming. However, only "and," "search," and "unification" parallelism can be used by the logic machines in the depth-first class. The amount of parallelism available to a machine in the dynamic class varies depending on the type of heuristic function being used.

REFERENCES

42. Tick, E. and D. Warren. "Towards a Pipelined Prolog Processor."

From the collection of the Computer History Museum (www.computerhistory.org)


