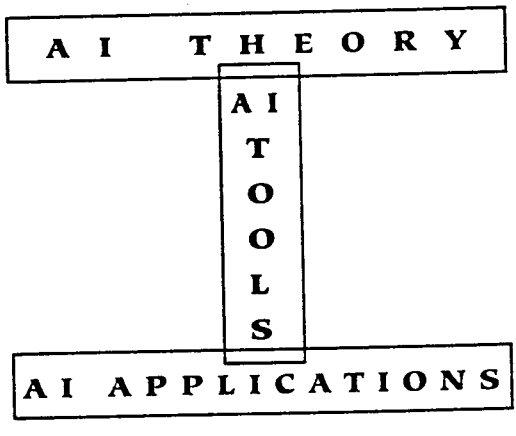
INTERNATIONAL JOURNAL ON

ARTIFICIAL INTELLIGENCE TOOLS

(Architectures, Languages, Algorithms)



VOLUME 3 NUMBER 1 MARCH 1994

Editor-in-Chief
NIKOLAOS G. BOURBAKIS
Co-Editor-in-Chief
JEFFREY J. P. TSAI

World Scientific
Singapore • New Jersey • London • Hong Kong

International Journal of Artificial Intelligence Tools, Vol. 3, No. 1 (1994) 97-125

AN ASSOCIATIVE DATA PARALLEL COMPILATION MODEL FOR TIGHT INTEGRATION OF HIGH PERFORMANCE KNOWLEDGE RETRIEVAL AND COMPUTING

ARVIND K. BANSAL
Department of Mathematics and Computer Science, Kent State University, Kent, OH 44242-0001, USA
E-mail: arvind@mcs.kent.edu

Received 18 May 1993
Revised 25 January 1994

ABSTRACT

Associative Computation is characterized by intertwining of search by content and data parallel computation. An algebra for associative computation is described. A compilation based model and a novel abstract machine for associative logic programming are presented. The model uses loose coupling of left hand side of the program, treated as data, and right hand side of the program, treated as low level code. This representation achieves efficiency by associative computation and data alignment during goal reduction and during execution of low level abstract instructions. Data alignment reduces the overhead of data movement. Novel schemes for associative manipulation of aliased uninstantiated variables, data parallel goal reduction in the presence multiple occurrences of the same variables in a goal. The architecture, behavior, and performance evaluation of the model are presented.

Keywords: Artificial intelligence: Associative computing; Data parallel computing; High performance; Knowledge bases: Knowledge retrieval; Logic programming

1. Introduction

Many problems such as modeling of aero-elasticity in aircrafts, image understanding systems, natural language understanding systems, geographic information systems, financial accounting systems, robot navigation, and genome sequencing require efficient integration of high performance intelligent reasoning, efficient information retrieval from large knowledge bases, and massive parallel scientific computing. In recent years, the logic programming paradigm [1, 2] has become a popular tool for AI systems and knowledge representation due to its natural capability to store databases, the declarative nature of programming, and the power of nondeterministic computation which facilitates alternate solutions to a problem. Previous approaches for high performance intelligent processing under logic programming

paradigm are as follows:

- Spawning and mapping concurrent processes either on MIMD architectures or pipelined vector computers [3, 4]: these models suffer from combinatorial explosion of processes. In process based models overhead of spawning processes is linearly proportional to the number of data elements. Such schemes are not suitable for data intensive problems which need same abstract computation on a large amount of data.
- Simulation of conventional abstract machines [5] on massive parallel machines: these models simulate pointer based data representation, and indexing based upon predicate name to identify procedures used for goal reduction. The use of pointer based data representation causes inherent sequentiality, and indexing scheme restricts the query power.
- Exploitation of associative search and data parallel computation [6, 7, 8]: these models do not exploit intertwining of search by content and data parallel computation.
- Loose coupling of intelligent reasoning on conventional architectures using conventional data representation and data parallel scientific computing on massively parallel supercomputers: these schemes suffer from data transfer overhead between architectures and run time overhead to transform data between conventional data representation schemes and data representation schemes on massively parallel supercomputers.

The implementations based upon conventional models, which use indexing on predicate-name to select a clause and pointer-based data representation, are incapable of answering a large class of queries needed in real world knowledge retrieval and data parallel computation. Some examples are

(a) queries which derive the set of objects based upon incomplete information about their attributes,
(b) queries which relate one or more objects without *a priori* knowledge of the relaionship,
(c) queries which reason about meta-relations — relations relating relations — about the objects, and
(d) queries integrating inequalities, ranges and data parallel computations.

The following two examples illustrate the above deficiencies: Example 1 illustrates the power of associative search to derive unspecified relations for the given attributes and handling of queries with meta-relations; Example 2 illustrates intertwining of search by content, inequality tests, and data parallel computation.

Example 1:

Conventional models answer queries such as "Who is a sister of Tom?"; "Who is a brother of Mary"?; "Who is a parent of Mary?"; "Who is a parent of John" etc

These models can not answer queries such as "How are Tom and Mary related?"; "How are Mary and John related?"; "Specify the relatives and their corresponding relationships to Tom?"; and "Who are relatives of Tom?". The corresponding query formulations are given in Table 1: the first column illustrates the class of queries, the second column illustrates formulation, the third column shows the query processing scheme, and the fourth column shows the processing capability of conventional systems.

```
% Meta relations

relative(sister). relative(brother). relative(parent).

% Concrete relations

brother(X, Y) :- parent(Z, X), parent(Z, Y), male(X), not_same(X, Y).

sister(X, Y) :- parent(Z, X), parent(Z, Y), female(X), not_same(X, Y).

parent(jane, mary).     parent(ram, mary).
parent(jane, tom).      parent(ram, john).

male(tom).      male(john)      male(ram).

female(mary)    female(jane).
```

Table 1. Query power extension in associative model.

GR: Goal reduction. FL: fact lookup, MR: meta relation

Class	Goal	Type	Con.
brother of mary?	*brother*(*mary*, *X*)?	GR	Yes
relationship of tom with mary?	*P*(*tom*, *mary*)?	FL	No
relationship of mary with tom ?	*P*(*mary*, *tom*)?	GR	No
relationships and relatives of tom?	*P*(*tom*, *X*)?	GR	No
relatives of tom?	*relative*(*P*) $\wedge$	MR	No
	P(*tom*, *X*)?	GR	

Example 2:

Consider a large data base of employees in a company. We have to identify the set of employees within the salary range $25{,}000 < salary < 35{,}000$ in a specific department "D". The salaries of these employees have to be raised by 5%. First the subset of employees is selected using associative search with inequality tests $salary > 25{,}000$ and $salary < 35{,}000$, and equality test $department =$ "D". After the selection of the subset, data parallel computation is used to compute the raise. This intertwining of associative search and data parallel computation is not possible in conventional systems.

In this paper, we present an algebra for associative computing, and use this algebra to develop a novel computation model of associative logic programming which achieves tight integration of associative knowledge retrieval, data parallel computation, and rule based programming without data transformation and data transfer overhead. The model introduces many novel implementation techniques as follows:

- Abstract data is represented to exploit seamless integration of associative search and data parallel computation [9, 10].
- Data parallel matches and data parallel computations are used to release bindings to reduce overhead of finding alternate solutions,
- A data parallel environment is used to facilitate associative computation on bags of bindings, vectors, sets, and sequences.
- Different components of the abstract machine are aligned to reduce the overhead of data transfer and data transformation.

The compiler and emulator for the proposed model have been implemented. The compiler has been written in C^{++}. The emulator has been written using ANSI C. The model is portable to any architecture which supports a data parallel version of C. The benchmark results show:

- Knowledge retrieval is independent of the number of ground facts.
- Knowledge retrieval is possible by incomplete information which makes knowledge discovery possible.
- Relations with a large number of arguments are handled efficiently with little overhead.
- Associative lookup speed is quite comparable to data parallel computations, which allows the tight integration of high performance knowledge retrieval and data parallel computation without any overhead of data movement and data transformation.
- The model is efficient for both scalar and data parallel computations on various abstract data such as sequences, matrices, bags, and sets.

Implications of these results are that the model can be successfully applied to data intensive problems such as geographical information systems, image understanding systems, statistical knowledge bases, and genome sequencing. For example, in geographical information systems, spatial data structures such as quad-trees and oct-trees are represented associatively; different regions having the same values can be identified using associative search on values [10], and integrated with intelligent rule based reasoning. Integration of data parallel scientific computing, knowledge base retrieval, and rule based reasoning provides necessary tool for image understanding systems. Statistical queries can directly benefit from associative search by content, associative representation of structures, data parallel arithmetic computations, and data parallel aggregate functions. Genome sequencing requires

integration of knowledge retrieval, efficient insertion and deletion of data elements, and efficient manipulation of matrices for heuristic matching of sequences.

Section 2 describes the associative computing paradigm and an algebra for associative computation. Section 3 discusses definitions of logic programming, Warren Abstract Machine (WAM) — the compilation model on conventional machines, and advantages of incorporating associative computing over conventional schemes. Section 4 describes an associative abstract machine. Section 5 describes the advantages achieved by seamless integration of associative search, data parallel computation, and data alignment in the abstract machine. Section 6 describes the model behavior. Section 7 discusses the performance evaluation and compares the work with other related works. The last section concludes the paper.

2. Background

In this section, the definitions and mathematical notations, associative architecture [10, 11], associative computing [10], and associative representation of abstract data such as matrix, tree, and quad-tree are presented.

2.1. Preliminary Definitions and Notations

A bag is a collection of items such that there can be multiple occurrences of an element. A *D-bag*, denoted by $\mathcal{D}$, is defined as an ordered bag which also contains null elements $\perp$. For example, $\{2, \perp, 3\}$ is a D-bag. However, $\{2, \perp, 3\} \neq \{\perp, 2, 3\}$ since D-bags are ordered. $\perp \preceq$ every element in the D-bag. A D-bag $\mathcal{D}_1 = \{d_{11}, \ldots, d_{1N}\}$ is D-included in another D-bag $\mathcal{D}_2 = \{d_2, \ldots, d_{2N}\}$ if $\forall I_{(1 \leq I \leq N)}\ d_{1I} \preceq d_{2I}$. For example, $\{4, \perp, 5, 6\}$ is a *D-subbag* of $\{4, 3, 5, 6\}$ since $\perp \preceq 3$. D-union of two D-subbags of a D-bag derives a new D-bag $\{d_{31}, \ldots, d_{3N}\}$ such that $\forall I_{(1 \leq I \leq N)}$ $d_{3I} = d_{1I}$ if $d_{2I} \preceq d_{1I}$, $d_{3I} = d_{2I}$ if $d_{1I} \preceq d_{2I}$, or $d_{3I} = d_{1I}$ if $d_{1I} = d_{2I}$. For example, D-union of the D-subbags $\{\perp, b, c\}$ and $\{a, b, \perp\}$ derives the D-bag $\{a, b, c\}$. D-intersection of two D-subbags of a D-bag derives a new D-bag such that $\forall I_{(1 \leq I \leq N)}$ $d_{3I} = d_{1I}$ if $d_{1I} \preceq d_{2I}$, $d_{3I} = d_{2I}$ if $d_{2I} \preceq d_{1I}$, or $d_{3I} = d_{1I}$ if $d_{1I} = d_{2I}$. For example, D-intersection of D-subbags $\{2, 3, \perp\}$ and $\{\perp, 3, 4\}$ derives the D-subbag $\{\perp, 3, \perp\}$. Note that D-inclusion, D-union, and D-intersection of two D-subbags are different than usual definitions of union and intersection since D-inclusion, D-union, and D-intersection are based on pairwise comparison of elements in two D-bags. The truth values *true* and *false* are treated synonymously with the values "1" and "0" respectively. An F-bag is a D-bag which has either *true* and *false*. We treat *false* (or "0") $\preceq$ *true* (or "1"). A F-bag of *1s* is denoted by $\mathcal{F}^1$, a F-bag of *0s* is denoted by $\mathcal{F}^0$, and a F-bag containing both *1s* and *0s* is denoted by $\mathcal{F}$. F-bags are realized by logical bit-vectors. Under the assumption *false* $\preceq$ *true*, D-union of F-bags and logical-OR of the corresponding logical bit-vectors are equivalent, and D-intersection of F-bags and logical-AND of the corresponding logical bit-vectors are equivalent. Cartesian product is denoted by $\times$, equivalence is denoted by $\equiv$, isomorphism is denoted by $\cong$, inclusion is denoted by $\subseteq$, D-inclusion

is denoted by $\sqsubseteq$, intersection of two bags is denoted by $\bigcap$, D-intersection of two D-bags is denoted by $\bigsqcap$, union of two bags is denoted by $\bigcup$, D-union of two D-bags is denoted by $\bigsqcup$, logical-AND is denoted by $\bigwedge$, logical-OR is denoted by $\bigvee$, underived value is denoted by $\top$, data parallel computation on D-bags is denoted by $\bigodot^D$, and computation on a pair of data element or a singleton data element is denoted by $\odot$, associative update is denoted by $\uplus$, associative deletion is denoted by $\ominus$, and communication cost between two data elements d_I and d_J is denoted by $e(d_I, d_J)$. Both $\bigodot$ and $\odot$ are generic representation of an operator. A subscript is used to denote a specific instance of an abstract data, and a superscript is used to denote a subclass.

2.2. Architecture for Associative Computing

An architecture is a sequence of *processing cells*. Each cell is a quadruple $< C_i, R_i, S_i, M_i >$ where C_i denotes a processing element (PE), R_i denotes a set of local registers, S_i denotes local storage, and M_i denotes a mask-bit. The D-bag of mask bits is set selectively to filter instructions. An instruction is broadcast to each cell simultaneously. The flow of control is effected by generating, saving, and restoring the mask bits based on the results of tests on local data [10, 11]. An associative search of a field for a specific value sets up the corresponding mask bit which is stored and manipulated during computations. SIMD architecture with content addressable memory [11] (see Fig. 1) satisfies this criteria. However, we do not limit the scope of the associative computation to SIMD architectures.

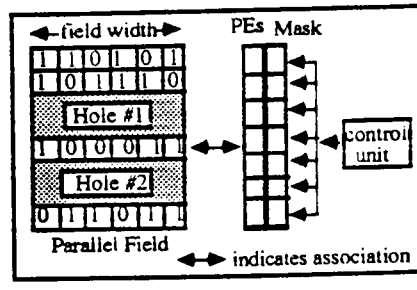


Fig. 1. SIMD Architecture and Parallel Field.

2.3. Associative Computing Paradigm

Associative computing [10] is characterized by a seamless integration of *data-parallel computation* [12], *association of data elements* [13], and *search by content* [11]. *Data-parallelism* is characterized by performing the same abstract computations concurrently on a bag of data [12].

Data parallel computations are independent of the number of elements in a D-bag since each data element can be mapped on a single PE in a unit time

For SIMD computers, the data parallel computation is performed in a lock-step fashion. However, the notion of data parallel computation is more general: same abstract computations on individual elements are performed concurrently without any control dependency. A computation is *data sequential* if the same abstract computations are performed individually on the elements, one element at a time, of the same bag. Partial data parallelism is exploited if M elements (M $\leq$ number of elements in the bag) are processed at a time. If the number of elements in a field is larger than the maximum number of processing elements, the elements are folded, and the processing time is multiplied by O(number of elements/number of processing elements).

2.4. Representing Data on Associative Architecture

Data items with the same data type are stored in a *parallel field* — a two dimensional memory organization of columns and rows with each cell attached to each row. Each row stores a data item as illustrated in Fig. 1. A D-bag is realized using a parallel field. $\perp$ is realized using a hole which means absence of an element. A F-bag is realized using *logical bit-vectors* as illustrated in Fig. 2. D-union of two F-bags is realized by logical-OR of the corresponding logical bit-vectors; D-intersection of two F-bags is realized by logical-AND of the corresponding logical bit-vectors. For example, $\{0, 1, 0\} \bigsqcup \{1, 0, 1\} \equiv \{0 \vee 1, 1 \vee 0, 0 \vee 1\} \equiv \{1, 1, 1\}$. Similarly, $\{1, 1, 0\} \bigsqcap \{0, 1, 0\} \equiv \{1 \wedge 0, 1 \wedge 1, 0 \wedge 0\} \equiv \{0, 1, 0\}$.

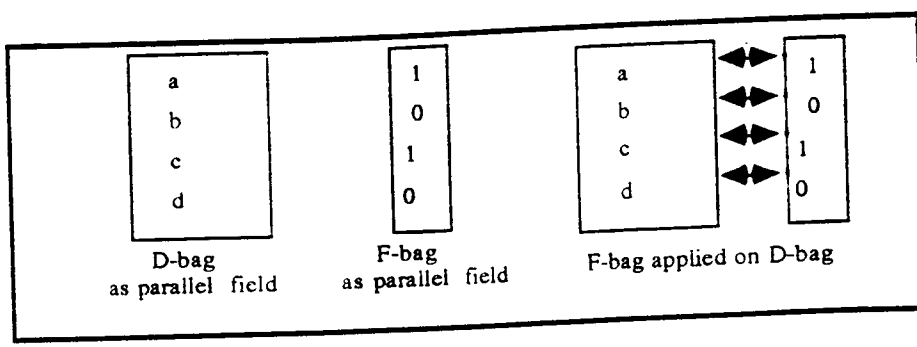


Fig. 2. Associative representation of data.

2.4.1. Notations Related to Associative Data

Number of D-bags are denoted by M (where $M \geq 1$), and number of elements in a D-bag are denoted by N (where $N \geq 1$). Using association of parallel fields, a D-bag of M-tuples of the form $\{< d_{11}, \ldots, d_{M1} >, \ldots, < d_{1N}, \ldots, d_{MN} >\}$ is stored as M aligned D-bags. M D-bags are aligned if the communication distance $\forall\, (I, J)_{1 \leq (I,J) \leq M)}\ e(d_{IK} \in \mathcal{D}_I,\ d_{JK} \in \mathcal{D}_J)$ is same for all tuples.

An association of M D-bags is denoted by $\mathcal{D}_1 \bigoplus \mathcal{D}_2 \bigoplus \cdots \bigoplus \mathcal{D}_M$ where $\bigoplus$ represents the association of two D-bags. A projection of a D-bag from an association of D-bags is denoted by Π_I, and selection of a specific element from a tuple is denoted by π_I. For example, $\Pi_{I(1 \leq I \leq M)}\ (\mathcal{D}_1 \bigoplus \mathcal{D}_2 \bigoplus \cdots \bigoplus \mathcal{D}_M) = \mathcal{D}_I$, and $\pi_{I(1 \leq I \leq N)}\ (< d_{11}, \ldots, d_{1N} >) = d_{1I}$. An association of a D-bag with an F-bag is denoted by $\bigotimes$. Association of a D-bag and an F-bag is different from an association of two D-bags since F-bags are applied on D-bags to select the corresponding elements.

2.5. Associative Representation of Abstract Data

A two-dimensional matrix is represented as a D-bag of triples of the form $\{(index_1^1, index_1^2, value_1), \ldots, (index_N^1, index_M^2, value_{NM})\}$. Associative representation is quite efficient for handling sparse matrices as only non-zero entries are stored. A tree is represented as a D-bag of the pairs of the form $\{(Path_1, Value_1), \ldots, (Path_I, Value_I), \ldots, (Path_N, Value_N)\}$ where N is the number of nodes in the tree; $Path_I$ is the path encoding of the node from the root node, and $Value_I$ is information stored at the node. For details of tree representation and manipulation, the reader may refer to Refs. 9 and 14. A quad-tree, an example of spatial data structure, is represented as a D-bag of 5-tuples of the form $\{(X\_lb_1, X\_ub_1, Y\_lb_1, Y\_ub_1, Value_1), \ldots, \{(X\_lb_I, X\_ub_I, Y\_lb_I, Y\_ub_I, Value_I), \ldots, \{(X\_lb_N, X\_ub_N, Y\_lb_N, Y\_ub_N, Value_N)\}$ where N is the number of nodes in a quad-tree. Where "lb" denotes lower bound, and "ub" denotes upper bound. Similarly, an oct-tree is represented a D-bag of 7-tuples of the form $\{(X\_lb_1, X\_ub_1, Y\_lb_1, Y\_ub_1, Z\_lb_1, Z\_ub_1, Value_1), \ldots, \{< X\_lb_I, X\_ub_I, Y\_lb_I, Y\_ub_I, Z\_lb_I, Z\_ub_I, Value_I >, \ldots, \{< X\_lb_N, X\_ub_N, Y\_lb_N, Y\_ub_N, Z\_lb_N, Z\_ub_N, Value_N >\}$. Any partial data structure satisfying a specific property is represented as a D-subbag $\mathcal{D} \bigotimes \mathcal{F}$ where $\mathcal{D}$ is the original D-bag and $\mathcal{F}$ is derived by searching for a specific attribute.

Example 3:

Consider the various abstract representation of data. For example, the sparse matrix in Fig. 3A is represented as $\{< 1, 1, 10 >, < 100, 50, 30 >, < 100, 90, 50 >\}$. The binary tree in Fig. 3B is represented as $\{< 100, a >, < 110, b >, < 120, c >, < 111, b >, < 112, d >\}$. The quad tree in Fig. 3C is represented as $\{< 0, 50, 0, 50, \text{"corn"} >, < 50, 75, 50, 75, \text{"wheat"} >, < 75, 100, 50, 75, \text{"corn"} >, < 75, 100, 75, 100, \text{"water"} >, < 50, 100, 0, 50, \text{"water"} >\}$.

3. Overview of Logic Programming Concepts

This section reviews definitions of logic programming [1, 2], describes WAM — the conventional implementation schemes of logic programs [5], and presents advantages of associative computation [6, 9, 14] over conventional computation schemes.

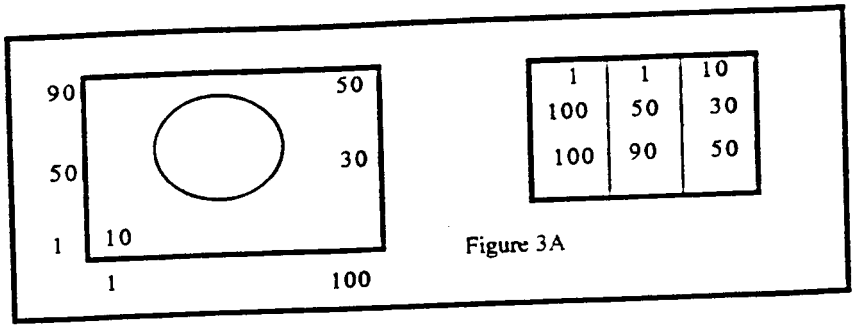
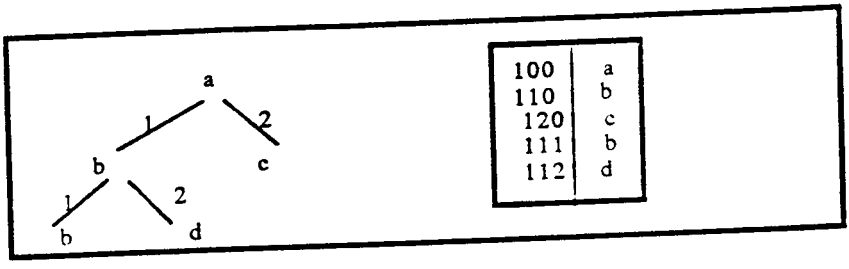
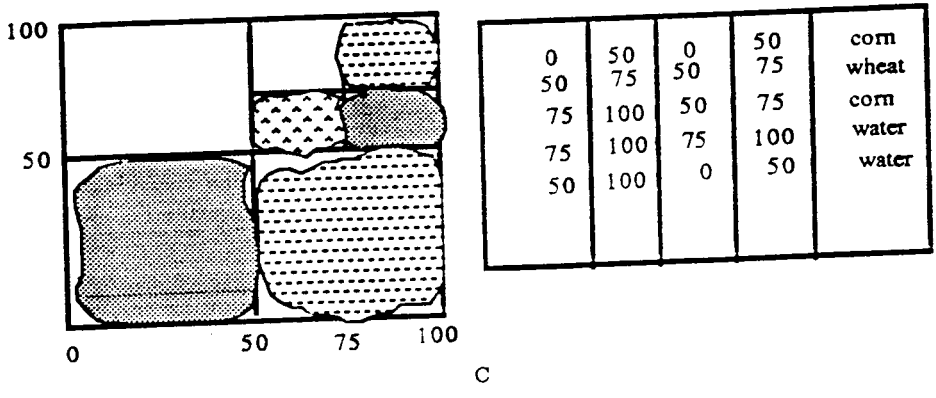


Figure 3A

A



B



C

Fig. 3. Associative representation of abstract data.

A *logic program* is a set of *facts* and *rules*. The set of facts and rules having the same name and number of arguments form a *procedure*. Each rule (Horn clause) is of the form $A :- B_1, \ldots, B_N$ $(N \geq 0)$. For a fact, N is 0. The left hand side of a rule is a *clause-head*, and each literal B_i on the right-hand side is a *subgoal*. While facts form the data part of a program, rules are used to reduce a query to simpler subqueries. The execution model is based upon reduction of a query to a conjunction of subqueries (subgoals) in a repeated manner. The heart of the reduction process

is *unification*: a pattern matching process, which is used for matching the values of the same type of data objects, associating a value with a variable, and passing the values between procedures. The process of unbinding caused by the failure of the unification between a goal term and a clause-head, and the selection of an alternate clause during a goal reduction is called *shallow backtracking*. The process of restoring the environment when a goal can not unify with any of the clause-heads in the given procedure is called *deep backtracking*.

Variables begin with a capital letter, and constants begin with a small letter. A *multiple-occurrence variable* occurs more than once in a literal. A *single-occurrence variable* occurs once in a literal. For example, in the literal $a(X, X, b, Y)$, the variable X is a multiple-occurrence variable, and the variable Y is a single-occurrence variable. A shared variable occurs in at least two subgoals. The first occurrence of a shared variable is called a *producer* and other occurrences of the variable are called *consumers*.

Two uninstantiated variables are aliased if they share the same value. Aliasing is an equivalence relationship: any variable V_1 is aliased to itself; if V_1 is aliased to V_2 and V_2 is aliased to V_3 then V_1 is alias to V_3; if the variable V_1 is aliased to V_2 then V_2 is aliased to V_1. Aliasing of any two variables $V_{I(1 \leq I \leq N)} \in$ alias set $\{V_1, V_2, \ldots, V_N\}$ and $W_{J(1 \leq J \leq M)} \in$ alias set $\{W_1, W_2, \ldots, W_M\}$ gives a new alias set $\{V_1, \ldots, V_N, W_1, \ldots, W_M\}$ which is derived by the union of two sets.

A fact is a *simple fact* if it has no aliased variables. A fact is a *complex fact* if it has at least one multiple-occurrence variable. A clause is a *complex clause* if it is either a complex fact or a clause with non-empty clause-body.

Meta relations treat relations as objects. In Example 1, the predicate *relative/1* is a meta relation. The positions of the objects is significant in a query. For example, "Who is Tom related to under what relationship?" and "Who are related to Tom under what relationship"? are expressed as "$P(tom, X)$?" and "$P(X, tom)$?" respectively. The uninstantiated variable P indicates unspecified relation names, and the variable X indicates unspecified objects.

3.1. WAM — Conventional Execution Models

The Warren Abstract Machine [5] and its variations are the most popular models to compile and execute logic programs on conventional machines. Briefly, a WAM consists of five major components, namely, a *low level abstract instruction set*, a *heap*, a *local stack*, a *trail stack*, and a set of *argument registers* (see Fig. 4). A heap is a global shared area for storing bindings and complex data structures. A local stack stores the information related to calling procedures and control threads (traditionally known as choice-points). A trail stack stores the bindings temporarily and is used to restore the environment by matching and removing the bindings caused by the failed clauses, during backtracking. The argument registers are used to store the pointers to the arguments (either on the heap or on the local stack) of the procedure being executed. The use of argument registers reduces the dat

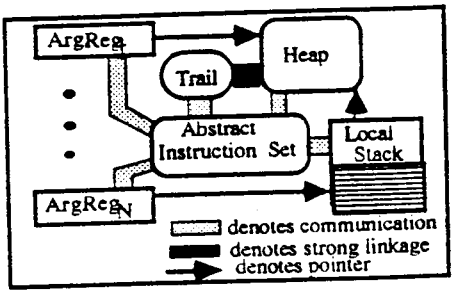


Fig. 4. WAM architecture.

transfer time significantly during a procedure call, and has the same application as the use of registers in assemblers. Storage of previous environment consists of argument registers to be altered and multiple pointers such as the pointer in the heap, pointer in the trail stack, pointer in the local stack. Previous environment is restored after successful execution of the called procedure or during backtracking. Complex data structures are stored and manipulated in the heap using chain of pointers. Allocation of these pointer-based structures cause run time overhead of memory allocation, and manipulation of data items is sequential due to traversal of chain of pointers.

3.2. Advantages of Associative Computation

The advantages of the associative computation model, exploited in the previous approaches [6, 9, 14, 15] are as follows:

(1) A goal position can be searched in one unit operation. An association of D-bags is used to hold all the elements of an argument in the set of clauses of a program. This process matches the ground values to identify non-unifiable clause-heads, and derives the bindings for the goal variables. The data parallel match reduces the shallow backtracking significantly. Figure 5 illustrates the concept. The goal $P(H, k)$ matches the whole set of clause-heads $\{a(b, k), a(c, k), a(d, c), \ldots, c(d, g), c(c, k\}$ in three unit data parallel matches.

(2) Bindings are associatively released in a constant number of operations during backtracking. Figure 6 illustrates the concept. The bindings related to time-stamp *20* are released in a unit data parallel match.

(3) Association of D-bags is used instead of conventional stacks to unify two logical terms. Use of associative search significantly reduces the overhead of searching for the bindings and the restoration of control thread.

(4) Data parallel computation is performed on all the elements of a D-bag simultaneously. In conventional schemes, such computations are dependent on the size of the input data, use tail recursion, and are at best close to linear iteration time.

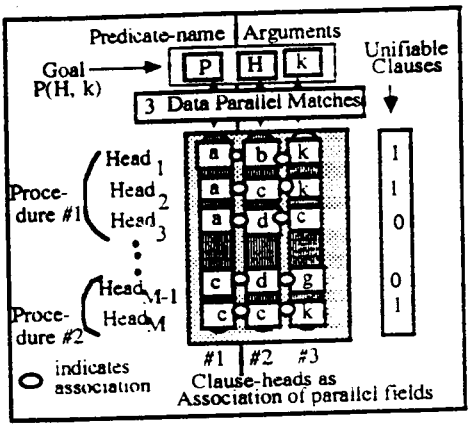


Fig. 5. Data parallel match in goal reduction.

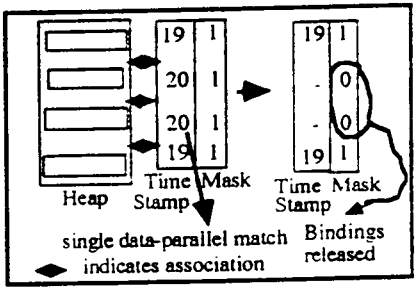


Fig. 6. Data parallel release during backtracking.

4. An Algebra of Associative Computation

This section describes an algebra for associative computation. This algebra is the basis of associative logic programming model. There are five types of laws: *laws for data association*, *laws for associative search*, *laws for associative selection*, *laws for data parallel computations* and *laws for associative update*. These laws have been used extensively in the development of the model. Each rule is denoted by $R^{A}_{I(1\leq I\leq 17)}$.

4.1. Laws of Data Association

The first rule states that a D-bag of M-tuples is given by the association of M D-bags such that corresponding elements in every tuple are aligned. For exampl

$\{a, 2, 3, 4\} \bigoplus \{b, 5, 6, 7\}$ is equivalent to $\{< a, b >, < 2, 5 >, < 3, 6 >, < 4, 7 >\}$.

$$\mathcal{D}_1 \bigoplus \cdots \bigoplus \mathcal{D}_M \equiv \{< d_{11}, \ldots, d_{M1} >, \ldots, < d_{1N}, \ldots, d_{MN} >\}$$ - Construction (4.1)

The second rule states that given a D-bag of tuples, a D-bag of elements can be projected in a unit operation.

$$\Pi_I(\mathcal{D}_1 \bigoplus \cdots \bigoplus \mathcal{D}_M) \Rightarrow \mathcal{D}_I.$$ - Projection (4.2)

The third rule states that resulting association is independent of level of sub-associations formed by individual D-bags. For example, $\{1, 2\} \bigoplus (\{3, 4\} \bigoplus \{5, 6\})$ is equivalent to $(\{1, 2\} \bigoplus \{3, 4\}) \bigoplus \{5, 6\}$, and both are equivalent to $\{< 1, 3, 5 >, < 2, 4, 6 >\}$.

$$\mathcal{D}_1 \bigoplus (\mathcal{D}_2 \bigoplus \mathcal{D}_3) \equiv (\mathcal{D}_1 \bigoplus \mathcal{D}_2) \bigoplus \mathcal{D}_3$$ - Associativity (4.3)

The fourth rule states that associations of D-bags are not symmetric. However, the associations are isomorphic: a pair $(x, y) \in \mathcal{S}_1 \bigoplus \mathcal{D}_2$ (such that $x \in \mathcal{D}_1$ and $y \in \mathcal{D}_2$) has a bijective mapping to $(y, x) \in \mathcal{D}_2 \bigoplus \mathcal{D}_1$. The implication of this rule is that the same information can be represented equivalently by permuting the order of association, and the communication distance between elements of a tuple is independent of the order of association.

$$(\mathcal{D}_1 \bigoplus \mathcal{D}_2 \not\equiv \mathcal{D}_2 \bigoplus \mathcal{D}_1) \bigwedge (\mathcal{D}_1 \bigoplus \mathcal{D}_2 \cong \mathcal{D}_2 \bigoplus \mathcal{D}_1)$$ - Antisymmetry (4.4)

4.2. Laws of Associative Search

The fifth rule states that associative search of a data element d in a D-bag $\mathcal{D}$ derives an F-bag $\mathcal{F}$ such that for every $d_j = d$, the corresponding element in F-bag is "1" otherwise the corresponding element in F-bag is "0". For example, associative search of an element 4 in the D-bag $\{3, 5, 4, 7, 4, 9\}$ gives an F-bag $\{0, 0, 1, 0, 1, 0\}$ and $\{3, 5, 4, 7, 4, 9\} \bigotimes \{0, 0, 1, 0, 1, 0\}$ derives $\{0, 0, 3, 0, 3, 0\}$. $d \in \mathcal{D} \Rightarrow \mathcal{D} \bigotimes \mathcal{F}$ such that

$$\forall I_{(1 \le I \le N)}\ (d = d_I) \Rightarrow (\pi_I(\mathcal{F}) = 1)$$
$$\forall I_{(1 \le I \le N)}\ (d \neq d_I) \Rightarrow (\pi_I(\mathcal{F}) = 0)$$ Membership (4.5)

The sixth rule states that by associatively searching in one field, the associated data elements in the other field can be extracted. For example, associative search for a tuple $\{4, \top, \top\}$ in a D-bag $\{< 4, 5, 6 >, < 3, 7, 9 >, \ldots, < 4, 9, 10 >\}$ derives the F-bag $\{1, 0, \ldots, 1\}$. The F-bag when applied on the D-bag selects

the D-subbag $\{< 4,5,6 >, \perp, \ldots, < 4,9,10 >\}$.

$$(\mathcal{D}_1 \oplus \mathcal{D}_2 \oplus, \ldots, \mathcal{D}_M) \bigwedge d \in \mathcal{D}_{J(1\leq J\leq M)} \equiv$$
$$(\mathcal{D}_1 \oplus \mathcal{D}_2 \oplus, \ldots, \mathcal{D}_M) \otimes \mathcal{F} \text{ such that}$$
$$\forall I_{(1\leq I\leq N)}\ ((\pi_J(< d_{1I}, d_{2I}, \ldots, d_{MI} >) = d) \Rightarrow \pi_I(\mathcal{F}) = 1)$$
$$\forall I_{(1\leq I\leq N)}\ ((\pi_J(< d_{1I}, d_{2I}, \ldots, d_{MI} >) \neq d) \Rightarrow \pi_I(\mathcal{F}) = 0). \qquad \text{Search (4.6)}$$

4.3. Laws of Associative Selection

The seventh rule states that association of an F-bag with a D-bag selects the data elements whenever the corresponding element in F-bag is 1. For example, $\{3,5,6\} \otimes \{0,1,0\} \equiv \{\perp, 5, \perp\}$.

$$\mathcal{D}_1 \otimes \mathcal{F} \Rightarrow \mathcal{D}_2 \text{ such that } (\mathcal{D}_2 \sqsubseteq \mathcal{D}_1) \text{ and}$$
$$\forall I_{(1\leq I\leq N)}\ (\pi_I(\mathcal{F} = 1)) \Rightarrow (d_{2I} = d_{1I})$$
$$\forall I_{(1\leq I\leq N)}\ (\pi_I(\mathcal{F} = 0)) \Rightarrow (d_{2I} = \perp). \qquad \text{- Selection (4.7)}$$

The eighth rule states that selecting data elements from two associated D-bags is equivalent to selecting data elements from individual D-bags and then associating the D-subbags. For example, $(\{4,5,6\} \otimes \{1,0,1\}) \oplus (\{a,b,c\} \otimes \{1,0,1\}) \equiv \{< 4, a >, < 5, b >, < 6, c >\} \otimes \{1,0,1\} \Rightarrow \{< 4, a >, \perp, < 6, c >\}$.

$$(\mathcal{D}_1 \oplus \mathcal{D}_2) \otimes \mathcal{F} \equiv (\mathcal{D}_1 \otimes \mathcal{F}) \oplus (\mathcal{D}_2 \otimes \mathcal{F}). \qquad \text{- Distributivity (4.8)}$$

The ninth rule states that data elements of a D-bag $\mathcal{D}$ selected by applying a F-bag $\mathcal{F}_1$ includes the data elements selected by applying another F-bag $\mathcal{F}_2$ if $\mathcal{F}_1 \sqsubseteq \mathcal{F}_2$. For example, $(\{5,6,7\} \otimes \{1,0,0\}) \equiv \{5, \perp, \perp\} \sqsubseteq (\{5,6,7\} \otimes \{1,0,1\}) \Rightarrow \{5, \perp, 6\}$

$$\mathcal{F}_1 \sqsubseteq \mathcal{F}_2 \Rightarrow (\mathcal{D} \otimes \mathcal{F}_1) \sqsubseteq (\mathcal{D} \otimes \mathcal{F}_2). \qquad \text{- Monotonicity (4.9)}$$

The tenth rule states that data elements of a D-bag selected by applying D-union of two F-bags is same as D-union of two D-subbags derived by applying individual F-bags on the D-bag. For example, $(\{2,3,4\} \otimes \{1,0,0\}) \bigsqcup (\{2,3,4\} \otimes \{0,1,0\}) \equiv \{2,3,4\} \otimes (\{1,0,0\} \bigsqcup \{0,1,0\}) \Rightarrow (\{2,3,4\} \otimes \{1,1,0\}) \Rightarrow (\{2,\perp,\perp\} \bigsqcup \{\perp,3,\perp\}) \Rightarrow < 2,3,\perp >$.

$$(\mathcal{D} \otimes \mathcal{F}_1) \bigsqcup (\mathcal{D} \otimes \mathcal{F}_2) \equiv \mathcal{D} \otimes (\mathcal{F}_1 \bigsqcup \mathcal{F}_2) \qquad \text{- D-union (4.10)}$$

The eleventh rule states that data elements of a D-bag selected by applying D-intersection of two F-bags is the same as D-intersection of two D-subbags derived by applying individual F-bag on the D-bag. For example, $(\{2,3,4\} \otimes \{1,1,0\}) \bigsqcap (\{2,3,4\} \otimes \{0,1,1\}) \equiv \{2,3,4\} \otimes (\{1,1,0\} \bigsqcap \{0,1,1\}) \Rightarrow (\{2,3,4\} \otimes$

$\{0, 1, 0\}) \equiv (\{2, 3, \perp\} \sqcap \{\perp, 3, 4\}) \Rightarrow \{\perp, 3, \perp\}$.

$$(\mathcal{D} \bigotimes \mathcal{F}_1) \sqcap (\mathcal{D} \bigotimes \mathcal{F}_2) \equiv \mathcal{D} \bigotimes (\mathcal{F}_1 \sqcap \mathcal{F}_2) \qquad \text{- D-intersection (4.11)}$$

The twelfth rule states that Cartesian product of a bag $\mathcal{D}$ with *true* is equivalent to applying $\mathcal{F}^1$ on $\mathcal{D}$, and is equivalent to $\mathcal{D}$ itself. For example, $\{2, 3, 4\} \times \{true\} \equiv \{2, 3, 4\} \bigotimes \{1, 1, 1\} \equiv \{2, 3, 4\}$.

$$\mathcal{D}_1 \times \{true\} \equiv \mathcal{D}_1 \bigotimes \mathcal{F}^1 \equiv \mathcal{D}. \qquad \text{- Identity (4.12)}$$

The thirteenth rule states Cartesian product of a D-bag $\mathcal{D}$ with *false* is equivalent to applying $\mathcal{F}^0$ on $\mathcal{D}$, and derives a null set $\emptyset$. A null set is also represented as a D-bag with every element as $\perp$.

$$\mathcal{D} \times \{false\} \equiv \mathcal{D} \bigotimes \mathcal{F} \equiv \emptyset. \qquad \text{- Zero (4.13)}$$

4.4. Laws of Data Parallel Computation

The fourteenth rule states that data parallel computation on two associated D-bags is equivalent to same computation on every pair of corresponding elements of the two D-bags. Any computation involving $\perp$ maps onto $\perp$. For example, $\{2, 3, \perp\} *^D \{3, 4, \perp\} \equiv \{2 * 3, 3 * 4, \perp * \perp\} \Rightarrow \{6, 12, \perp\}$.

$$(\mathcal{D}_1 \bigoplus \mathcal{D}_2) \bigwedge (\mathcal{D}_1 \bigodot^D \mathcal{D}_2) \Rightarrow$$
$$\{(d_{11} \odot d_{21}), \ldots, (d_{1N} \odot d_{2N})\} \qquad \text{- Vector-vector} \Rightarrow \text{vector (4.14)}$$

The fifteenth rule states that a data parallel computation involving a scalar value d and a D-bag $\mathcal{D}$ is equivalent to taking Cartesian product of the singleton set $\{d\}$ with $\mathcal{F}^1$, and performing data parallel computation on the association $(\{d\} \times \mathcal{F}^1) \bigoplus \mathcal{D}_1$. For example, $4 *^D \{2, 3, 4\} \equiv (\{4\} \times \{1, 1, 1\}) *^D \{2, 3, 4\} \equiv \{4, 4, 4\} *^D \{2, 3, 4\} \Rightarrow \{8, 12, 16\}$.

$$d \bigodot^D \mathcal{D} \equiv ((\{d\} \times \mathcal{F}^1) \bigoplus \mathcal{D}) \bigwedge ((\{d\} \times \mathcal{F}^1) \bigodot^D \mathcal{D}) \Rightarrow$$
$$\{(d \odot d_{11}), \ldots, (d \odot d_{1N})\}. \qquad \text{- Scalar-vector} \Rightarrow \text{vector (4.15)}$$

4.5. Laws of Associative Update

The sixteenth rule concerns *associative update*. The rule states that insertion of a tuple $< d_1, \ldots, d_M >$ in an association $\mathcal{D}_1 \bigoplus \cdots \bigoplus \mathcal{D}_M$, derives $(\mathcal{D}_1^U \bigoplus \cdots \bigoplus \mathcal{D}_M^U)$ where $\mathcal{D}_I^U$ denotes the updated D-bag. Each $\mathcal{D}_I^U$ is equal to $\mathcal{D}_I \bigcup \{d_I\}$. During insertion alignment is preserved. However, the position of insertion need not be fixed. For example, $< a, b, c > \uplus \{< 2, 3 > < 4, 5 >, < 6, 7 >\}$ derives $\{< 2, 3, a > < 4, 5, b >, < 6, 7, c >\}$.

$$(< d_1, \ldots, d_M > \uplus (\mathcal{D}_1 \oplus \cdots \oplus \mathcal{D}_M)) \equiv$$
$$(\mathcal{D}_1 \cup \{d_1\}) \oplus \cdots \oplus (\mathcal{D}_M \cup \{d_M\}) \wedge$$
$$\exists K_{(1 \le K \le N+1)} \; (\pi_K((\mathcal{D}_1^U \oplus \cdots \oplus \mathcal{D}_M^U) = < d_1, \ldots, d_M >) \wedge$$
$$\pi_K \; ((\mathcal{D}_1^U \oplus \cdots \oplus \mathcal{D}_M^U) \notin (\mathcal{D}_1 \oplus \cdots \oplus \mathcal{D}_M) \quad \text{- Aligned update (4.16)}$$

The seventeenth rule concerns *associative release*. The rule states that by associatively searching in one field, the associated data elements in the other field can be released by applying the complement of the F-bag derived during associative search (R_5^A). For example, associative release of a tuple {4, ⊤, ⊤} from a D-bag $\mathcal{D} = \{< 4, 5, 6 >, < 3, 7, 9 >, \ldots, < 4, 9, 10 >\}$ is derived by a sequence of computations: membership of the tuple in $\mathcal{D}$ derives a F-bag $\mathcal{F} = \{1, 0, \ldots, 1\}$. Complement of $\mathcal{F}$ derives $\neg \mathcal{F} = \{0, 1, \ldots, 0\}$. $\mathcal{D} \otimes \neg \mathcal{F}$ derives the D-subbag $\{\perp, < 3, 7, 9 >, \ldots, \perp\}$.

$$(< d_1, \ldots, d_M > \ominus (\mathcal{D}_1 \oplus \cdots \oplus \mathcal{D}_M)) \equiv$$
$$(\mathcal{D}_1 \oplus \mathcal{D}_2 \oplus, \ldots, \mathcal{D}_M) \otimes \neg \mathcal{F} \text{ such that}$$
$$\forall I_{(1 \le I \le N)} \; ((\pi_J(< d_{1I}, d_{2I}, \ldots, d_{MI} >) = d) \Rightarrow \pi_I(\mathcal{F}) = 1)$$
$$\forall I_{(1 \le I \le N)} \; ((\pi_J(< d_{1I}, d_{2I}, \ldots, d_{MI} >) \neq d) \Rightarrow \pi_I(\mathcal{F}) = 0). \quad \text{Deletion (4.17)}$$

5. Structure of the Model

The compilation model maps a program, denoted by $\mathcal{P}$, as a pair of associations of the form $< \mathcal{L} \oplus \mathcal{P}^N \oplus \mathcal{A}_1 \oplus \cdots \oplus \mathcal{A}_N, \mathcal{L} \oplus \mathcal{P}^C >$. The first element of the pair, denoted by $\mathcal{P}^H$, represents a D-bag of clause-head tuples, and second element of the pair, denoted by $\mathcal{P}^B$, represents a D-bag of clause-body tuples. $\mathcal{L}$ is a D-bag of labels connecting clause-heads to low level code of the corresponding clause-body, $\mathcal{P}^N$ is a D-bag of procedure-names, $\mathcal{A}_I$ is a D-bag of I_{th} argument in set of the clause-heads in a program, $\mathcal{P}^C$ is a D-bag of a sequence of compiled instructions corresponding to the set of clause-bodies in the program such that each element $c_i \in \mathcal{P}^C$ is a sequence of instructions corresponding to one clause-body. A schematic of program representation is given in Fig. 7.

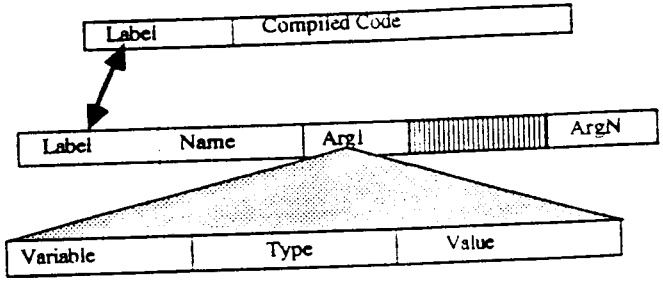


Fig. 7. Associative program representation.

The abstract machine architecture is a 7-tuple of the form $< \mathcal{I}, \mathcal{E}^D, \mathcal{E}^G, \mathcal{E}^V, \mathcal{E}^H, \mathcal{E}^S, \mathcal{R} >$. $\mathcal{I}$ is a set of abstract instructions; $\mathcal{E}^D$ is a data parallel environment which holds the D-bag of bindings for a non-scalar variable; $\mathcal{E}^H$ is a heap which holds the scalar bindings for variables and references of bag of bindings for variables with multiple values; $\mathcal{E}^H$ and $\mathcal{E}^V$ are the associations used to handle aliasing of uninstantiated variables; $\mathcal{R}$ is a bag of global registers to store temporary values; and $\mathcal{E}^S$ is a control stack which is used to store the control thread during program execution. The abstract instruction set consists of instructions for associative matching of the goal arguments to the corresponding clause heads, binding a goal variable to a D-bag of values, extracting a scalar value from a D-bag binding, D-intersection and D-union of F-bags, testing and backtracking for null F-bag representing unifiable clauses, shallow backtracking, data movement between the heap and the global registers, and procedure calls. The details of individual instructions and their operational semantics are outside the scope of this paper, and are given in Refs. 16 and 17. The individual components of the architecture are aligned to make use of the associative computing as detailed in Sec. 4. The overall scheme to execute a logic program is a tuple $< \mathcal{P}^H \oplus E^D, \mathcal{P}^B, \mathcal{E}^G \oplus \mathcal{A}^V \oplus \mathcal{R}, \mathcal{E}^S, \mathcal{E}^H >$ as demonstrated in Fig. 8.

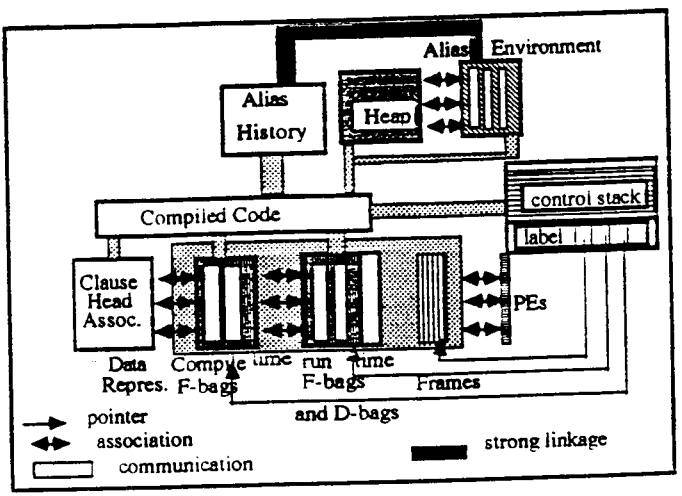


Fig. 8. Associative abstract machine with data alignment.

5.1. Associative Representation of Clause-heads

The association of clause-heads is represented as $\mathcal{L} \oplus \mathcal{P}^N \oplus \mathcal{A}_1 \oplus \cdots \oplus \mathcal{A}_M$. Each argument itself is an association of the form $\mathcal{V} \oplus \mathcal{B}^D \oplus \mathcal{B}^V$ where $\mathcal{V}$ is a D-bag of variables, $\mathcal{B}^D$ is a D-bag of binding type (integer, real, atom etc.), and $\mathcal{B}^V$ is a D-bag of actual values of the variables. A clause-head is treated as

$(M + 2)$ tuple of the form $< \pi_I(\mathcal{L}), \pi_I(\mathcal{P}^N), \pi_I(\mathcal{A}_1), \ldots, \pi_I(\mathcal{A}_M) >$ where $\pi_I(\mathcal{L})$ is connects clause-head and the clause-body, $\pi_I(\mathcal{P}^N)$ is the name of the literal, and $\pi_I(\mathcal{A}_{J(1 \leq J \leq M)})$ is an argument in the clause-head.

5.2. Associative Representation of Global Bindings

A heap, denoted by $\mathcal{E}^G$, is an association $\mathcal{T}^1 \oplus \mathcal{T}^2 \oplus \mathcal{V} \oplus \mathcal{B}^B \oplus \mathcal{B}^D \oplus \mathcal{B}^V \oplus \mathcal{N}^D \oplus \mathcal{F}^R$. $\mathcal{T}^1$ denotes a D-bag of time-stamps when variables were created. $\mathcal{T}^2$ denotes a D-bag of time-stamps when variables were bound. $\mathcal{V}$ denotes a D-bag of variable-id. $\mathcal{B}^B$ denotes a D-bag of the binding type (singleton or D-bag). $\mathcal{B}^D$ denotes a D-bag of data type of the values in the bindings. $\mathcal{B}^V$ denotes a D-bag of the scalar binding of variables. $\mathcal{N}^D$ denotes a D-bag of the index of bindings stored in the data parallel environment. $\mathcal{F}^R$ is a F-bag used to mark the shared variables.

5.3. Storing and Manipulating Bags of Bindings

Data parallel environment, denoted by $\mathcal{E}^D$, stores D-bags, F-bags, and association of D-bags. D-bags are used to store the bindings of a multi-valued variable. $\mathcal{E}^D$ is a stack with two major differences from conventional stacks: each data element is a D-bag, F-bag, or their association. Each bag is aligned with processing elements to reduce overhead of data movement during data parallel computation. It is due to this data parallel environment that *vector–vector* $\rightarrow$ *vector* and *vector–scalar* $\rightarrow$ *vector* computations are independent of number of data elements.

5.4. Holding Temporary Values — A Data Parallel Version

Global registers, denoted by $\mathcal{E}^R$, are exact replica of heap except that global registers use the large set of registers present in PE's: each register keeps a tuple. Once all the registers in PEs are used then an image of these registers is stored in the data parallel binding environment. In order to reduce data movement overhead, previous frames of registers are aligned to PEs. The information about each frame of registers is stored in the control stack, and is restored during backtracking. Usage of large number of registers in a frame provides *persistence* which reduces the overhead of storage and retrieval of temporary values during procedure calls and backtracking.

5.5. Handling Control Flow

Control stack, denoted by $\mathcal{E}^S$, is an association $\mathcal{T} \oplus \mathcal{W} \oplus \mathcal{N}^C \oplus \mathcal{N}^D \oplus \mathcal{N}^A \oplus \mathcal{L}^B \oplus \mathcal{L}^R \oplus \mathcal{L}^C$. $\mathcal{T}$ is a D-bag of time-stamps needed to restore previous environment. During backtracking, the current time-stamp is decremented, the corresponding tuple in $\mathcal{E}^S$ is searched using the new value of time-stamp, and the the environment is restored. $\mathcal{W}$ is a D-bag of index of frames of the global registers. $\mathcal{N}^C$ is a D-bag of index of the base-element of the bags generated at compile time. $\mathcal{N}^D$ is a D-bag of index of the base-element of the data parallel bindings, generated at run time, for a goal. $\mathcal{L}^B$ is a D-bag of labels of the code where the control has

to jump during backtracking. $\mathcal{L}^R$ is a D-bag of the labels of the code where the control has to jump to fetch next scalar value from a producer. $\mathcal{L}^C$ is a D-bag of labels of the code where control will jump after the successful execution.

5.6. Handling Aliasing

Aliased variables are handled associatively using two D-bags, namely, *alias set environment*, denoted by $\mathcal{E}^V$, and *alias history*, denoted by $\mathcal{E}^H$. $\mathcal{E}^V$ is a sequence of F-bags which are used to select different sets of aliased variables. $\mathcal{E}^V$ is aligned with $\mathcal{E}^G$ to derive the attributes of aliased variables. $\mathcal{E}^H$ is a D-bag of the form $\mathcal{T} \oplus \mathcal{N}^O \oplus \mathcal{N}^U$, and is used to restore the previous aliasing information during backtracking. $\mathcal{T}$ denotes a D-bag of time-stamps when new alias set was created. $\mathcal{N}^O$ denotes the D-bag of *F-bag index* (in $\mathcal{E}^V$) representing previous alias set of a variable, and $\mathcal{N}^U$ denotes the D-bag of *F-bag index* (in $\mathcal{E}^V$) representing current alias set of a variable.

6. Integrating Retrieval and Computation

This section describes the advantages achieved by alignment of D-bags and F-bags to reduce the overhead during goal reduction, retrieval of information from aligned fields, and data movement.

6.1. Alignment and Data Parallel Computation

Alignment of the arguments in the same clause facilitates

(i) data parallel pruning of those clauses which do not share the same values for multiple occurrence goal variables: arguments of the same clauses are pairwise equated in a data parallel manner. For example, data parallel equality on first two arguments for a goal $a(X, X, Y)$ and a set of clauses $\{a(5,6,4), \ldots, a(4,4,2), \ldots, a(6,7,9)\}$, derives a F-bag $\{0, \ldots, 1, \ldots, 0\}$ "(4.14)" indicating non-unifiability of clause-heads $a(5, 6, 4)$ and $a(6, 7, 9)$ "(4.7)". Figure 9 illustrates the implementation. Data parallel equality on $\mathcal{A}_I = \{a, b, c\}$ and $\mathcal{A}_J = \{b, b, c\}$ gives a F-bag $\{0, 1, 1\}$.

(ii) data parallel computation on two arguments of the same clause. For example, arithmetic comparison to identify whether second argument is greater than the third argument "(4.14)" for a goal $a(X, X, Z) \bigwedge X > Z$ and a set of clauses $\{a(5,6,4), \ldots, a(4,4,2), \ldots, a(6,7,9)\}$, a unit data parallel inequality test derives the F-bag $\{1, \ldots, 1, \ldots, 0\}$ which selects the clause $a(6, 7, 9)$ "(4.7)".

(iii) data parallel derivation of clauses satisfying complex conditions. For example, finding out the set of clauses which satisfy the goal $a(X, X, Z) \bigwedge X > Z$ will need D-intersection of two F-bags $\{0, \ldots, 1, \ldots, 0\}$ and $\{1, \ldots, 1, \ldots, 0\}$. Using "(4.10)" F-bag $\{0, \ldots, 1, \ldots, 0\}$ is derived, and using "(4.7)" the unifiable clause $a(4, 4, 2)$ is selected.

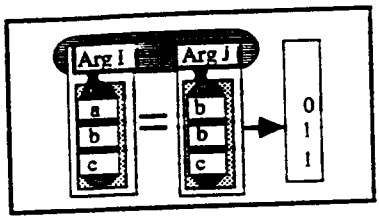


Fig. 9. Handling multiple occurrence goal variables.

6.2. Alignment and Data Movement

In conjunction with alignment of the program association with data parallel environment, the alignment of the arguments in a clause facilitates

(iv) selection of a subset of data in $\mathcal{P}^H$ without actual movement of data "(4.7)" and "(4.8)". For example, given a set of clauses $\{a(5,6,4), \ldots, a(4,4,2), \ldots, a(6,7,9)\}$, and an F-bag $\{1, \ldots, 1, \ldots, 0\}$, the arguments of clauses $a(5, 6, 4)$ and $a(4, 4, 2)$ are selected for data parallel computation without data movement.

(v) movement of selected data from $\mathcal{P}^H$ to $\mathcal{E}^D$ using a constant number of data parallel computations. For example, given a set of clauses $\mathcal{D} = \{a(5,6,4), \ldots, a(4,4,2), \ldots, a(6,7,9)\}$ and a F-bag $\mathcal{B} = \{1, \ldots, 1, \ldots, 0\}$, $\Pi_3(\mathcal{S} \otimes \mathcal{B})$ derives $\{4, 2, \perp\}$ in a unit time.

6.3. Alignment and Aliasing

One of the concerns in logic programs is to efficiently access and manipulate the aliased variables such that binding of one variable is also seen by the other aliased variable. Conventional systems use chain of references to access the bindings. In contrast, this model benefits by associative search (see Eq. (4.5)) to derive for the bindings of aliased variables; $\mathcal{V} \otimes \mathcal{F}$ is used to select an alias set and derive corresponding attributes in $\mathcal{E}^G$ using "(4.7)" and "(4.8)"; D-union of two F-bags is used to derive a new alias set "(4.10)"; bindings of all variables in an alias set are updated in constant number of data parallel computations using "(4.16)"; and bindings are accessed using "(4.5)" and "(4.6)".

6.4. Alignment and Associative Goal Reduction

There are four rules for associative goal reduction, namely, *matching constant pairwise data parallel equality*, *conjunctive pruning*, and *unification*. We use $\mathcal{F}_T^U$ as F-bag of unifiable clauses at time stamp T.

The first rule of matching constants states that matching the constant goal argument Arg_I^G with the corresponding D-bag $\mathcal{A}_I$ derives an F-bag $\pi_I(\mathcal{F}_T^U)$ (see Eq. (4.5)) which is used to prune those clauses which do not match with Arg_I^G using "(4.7)".

$((\text{type}(Arg_I^G) = \text{"constant"}) \bigwedge (Arg_I^G \in \mathcal{A}_I)) \Rightarrow (\mathcal{P}^H \bigotimes \Pi_I(\mathcal{F}_T^U))$ such that
$\text{type}(\pi_{I(1\leq I\leq N)}(\mathcal{A}_I)) \in \{\text{"constant"}, \text{"variable"}\}) \Rightarrow$

$$\pi_{I(1\leq I\leq N)}\ (\pi_{J(1\leq J\leq M)}(\mathcal{F}_T^U)) = \text{"1"}$$

$\text{type}(\pi_{I(1\leq I\leq N)}(\mathcal{A}_I)) \notin \{\text{"constant"}, \text{"variable"}\}) \Rightarrow$

$$\pi_{I(1\leq I\leq N)}\ (\pi_{J(1\leq J\leq M)}(\mathcal{F}_T^U)) = \text{"0"}. \qquad \text{Matching constants (5.1)}$$

The second rule of *data parallel equality* states that if two goal arguments Arg_I^G and Arg_J^G share the same multiple-occurrence variable, then non-unifiable clauses are pruned by equating the D-bags $\mathcal{A}_I$ and $\mathcal{A}_J$ (rule R_{13}^A).

$(type(Arg_I^G) = \text{"variable"}) \bigwedge (type(Arg_J^G) = \text{"variable"}) \bigwedge$

$(Arg_I^G = Arg_J^G) \equiv \mathcal{P}^H \bigotimes \Pi_{(I,J)(1\leq(I,J)\leq M)}(\mathcal{F}_T^U)$ such that

$\Pi_{(I,J)(1\leq(I,J)\leq M)}(\mathcal{F}_T^U) = (\mathcal{A}_I =^{\mathbf{D}} \mathcal{A}_J)$ - Data parallel equality (5.2)

The third rule of *conjunctive pruning* combines rules R_{10}^A and R_1^L and R_2^L to prune all the clauses which do not unify with the given goal. The rule takes intersection of every F-bag generated by using R_1^L and R_2^L and uses rule R_{10}^A to derive D-intersection.

$$\mathcal{F}_T^U = \Pi_1(\mathcal{F}_T^U) \sqcap \cdots \sqcap \Pi_M(\mathcal{F}_T^U) \qquad \text{- Pruning non-unifiable clauses (5.3)}$$

The fourth rule of unification is used for matching remaining goal arguments, and is generally used to handle aliased variables.

$\exists\ \mathrm{K}_{(1\leq K\leq N)}\ (\pi_K(\mathcal{F}_T^U) = \text{'true'}) \Rightarrow$

$\forall\ \mathrm{J}_{(1\leq J\leq M)}\ ((type(\pi_J(\pi_K(\mathcal{A}_1 \bigoplus \cdots \bigoplus \mathcal{A}_M))) = \text{"variable"}) \Rightarrow$

$\text{unify}\ (\pi_J(\pi_K(\mathcal{A}_1 \bigoplus \cdots \bigoplus \mathcal{A}_M)), Arg_J^G))$ - Partial unification "(5.4)"

6.5. Deriving Unspecified Relations for Objects

Given a query of the form $P(\mathcal{A}_1^G, \ldots, \mathcal{A}_N^G)$, any of the $N + 1$ positions may be uninstantiated. Only those queries where predicate variable P occurs as one of the arguments can not be answered. The query where all $(N + 1)$ elements are variables is meaningless. The query about unspecified predicate-names is processed in the same manner as to the queries with the instantiated predicate names. However, there is one major difference: the presence of an uninstantiated variable in the

position of predicate names may instantiate the variable to multiple predicate name. Since different predicate names represent different procedures, multiple procedures (with the same number of arguments) have to be handled in contrast to the case when the predicate names are instantiated.

7. The Model Behavior

The model exploits run time execution efficiency both at the data level during data parallel goal reduction by treating the clause-heads as association of D-bags for efficient pattern-matching, and control level during the execution of the code of the corresponding subgoals in the selected clause. The forward control flow is divided into three parts: *pre-call processing*, *pre-clause processing*, and *clause processing*.

Pre-call processing is used to perform data parallel goal reduction, and setting up the bindings for goal variables. Pre-call processing uses "(5.1)", "(5.2)", and "(5.3)" to derive unifiable clauses.

Pre-clause processing is used to test the presence of unifiable clauses in $\mathcal{F}_T^U$, and pass control to the right clause. Pre-clause processing is used to derive the potential bindings for a goal variable using R_4^L, and to jump to the code area associated with unifiable clause. If $\mathcal{F}_T^U$ is non-empty then next tuple $\pi_K(\mathcal{L} \oplus \mathcal{P}^N \oplus \mathcal{A}_1 \oplus \cdots \oplus \mathcal{A}_M)$ is selected in a non-deterministic manner, the corresponding clause-label $\pi_K(\mathcal{L})$ is picked, and control jumps to the corresponding instructions sharing the label (see Fig. 10). The order of selection of procedures after data parallel goal reduction is non-deterministic. Within a procedure, the order of selection of clauses within a procedure is non-deterministic.

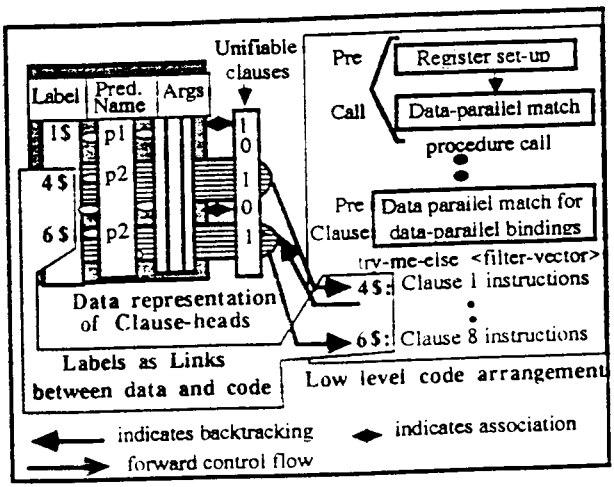


Fig. 10. Model behavior.

position of predicate names may instantiate the variable to multiple predicate name. Since different predicate names represent different procedures, multiple procedures (with the same number of arguments) have to be handled in contrast to the case when the predicate names are instantiated.

7. The Model Behavior

The model exploits run time execution efficiency both at the data level during data parallel goal reduction by treating the clause-heads as association of D-bags for efficient pattern-matching, and control level during the execution of the code of the corresponding subgoals in the selected clause. The forward control flow is divided into three parts: *pre-call processing*, *pre-clause processing*, and *clause processing*.

Pre-call processing is used to perform data parallel goal reduction, and setting up the bindings for goal variables. Pre-call processing uses "(5.1)", "(5.2)", and "(5.3)" to derive unifiable clauses.

Pre-clause processing is used to test the presence of unifiable clauses in $\mathcal{F}_T^U$, and pass control to the right clause. Pre-clause processing is used to derive the potential bindings for a goal variable using R_4^L, and to jump to the code area associated with unifiable clause. If $\mathcal{F}_T^U$ is non-empty then next tuple $\pi_K(\mathcal{L} \oplus \mathcal{P}^N \oplus \mathcal{A}_1 \oplus \cdots \oplus \mathcal{A}_M)$ is selected in a non-deterministic manner, the corresponding clause-label $\pi_K(\mathcal{L})$ is picked, and control jumps to the corresponding instructions sharing the label (see Fig. 10). The order of selection of procedures after data parallel goal reduction is non-deterministic. Within a procedure, the order of selection of clauses within a procedure is non-deterministic.

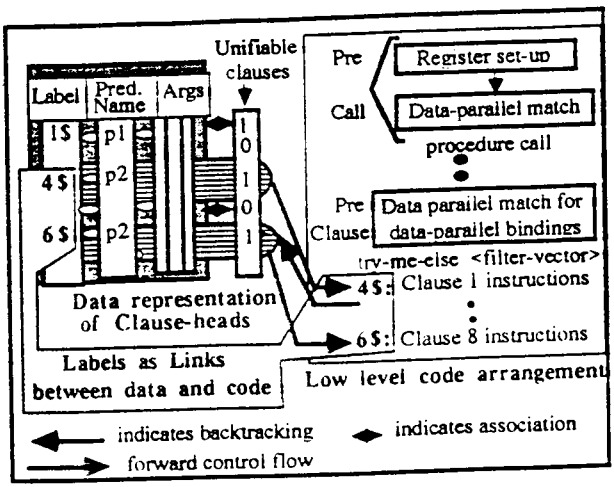


Fig. 10. Model behavior.

Clause processing is used to handle aliasing of variables, setting up the global registers, handling alternate bindings of shared variables during backtracking, and storing the current state into control stack, before starting next cycle.

During forward control flow, the global registers corresponding to the subgoal arguments are set, the time-stamp $\mathcal{T}$ is incremented by one, the old environment is stored in $\mathcal{E}^S$, and the pre-call processing is invoked. If at any time $\mathcal{F}_T^U = \emptyset$, deep backtracking occurs, information is retrive from the control stack, and the previous environment is restored. Upon backtracking, the old time-stamp $\mathcal{T}^O$ is picked up from the global register; the previous environments from the control stack are released using "(4.17)", and the data parallel environment is also restored back by setting the corresponding pointers; $\mathcal{E}^H$ and $\mathcal{E}^V$ are also restored. After deep backtracking, the control may jump back either to pre-clause processing area where alternate clause is selected for processing, or to clause area to pick up the next scalar value for a producer. The exact nature of backtracking is dependent upon the entry in $\mathcal{E}^S$.

Example 4

For example, in Fig. 10, after the data parallel goal reduction of a goal with an uninstantiated variable in the place of the predicate-name, three clauses with label 1$ in the procedure with predicate name *p1*, and labels 4$, and 6$ in the procedure with the predicate-name *p2* are selected. One of the procedures *p2* is selected, the control jumps to the code area starting with one of the labels 4$, and the corresponding instructions are executed. Upon backtracking, another label 6$ (in the same procedure) is selected, and control jumps to the code area starting with the label 6$. Upon the failure of the code with label 6$, the control jumps to the code area in the second procedure, and the instructions starting with the label 1$ are executed.

7.1. Handling Aliased Variables

The process of aliasing two variables V_1 and V_2 involves four steps as follows:

(i) The current F-bags $\mathcal{F}_1$ and $\mathcal{F}_2$ corresponding to alias sets of V_1 and V_2 are identified. This information is derived using associative search in $\mathcal{E}^G$, and the F-bag indices are picked from the D-bag $\mathcal{N}$ in $\mathcal{E}^G$ using "(4.6)".

(ii) D-union of alias sets represented by F-bags $\mathcal{F}_1$ and $\mathcal{F}_2$ (see Eq. (4.10)) is used to generate a new alias set selected by F-bag $\mathcal{F}_3$ which is stored in $\mathcal{E}^V$.

(iii) Two new entries are inserted in $\mathcal{E}^H$ to store the triple (time-stamp of creation of new alias set, index of previous alias set, index of current alias set) using "(4.16)". This history is needed to restore the environment during backtracking.

(iv) The index of the current alias set for each variable is updated in $\mathcal{E}^G$ using "(4.16)".

Instantiating any of the variables, automatically instantiates all the aliased variables by using an associate search (see Eq. (4.5)) and associative update (see Eq. (4.16)) in constant number of associative computations. Accessing the binding for aliased variables needs no extra access cost under this scheme "(4.5)" and "(4.6)". During backtracking, previous alias sets are recovered as follows:

(i) The indices for the current alias sets are identified by searching $\mathcal{T}$ (in $\mathcal{E}^H$) for the current value of time-stamp using "(4.6)".
(ii) The indices of old alias sets are selected from $\mathcal{N}^O$ in $\mathcal{E}^H$ using associative search on $\mathcal{N}^U$ using "(4.6)".
(iii) Data parallel update (see Eq. (4.16)) is used to replace the alias set index by the indices of old alias sets for each variable in the alias set.
(iv) The entries from $\mathcal{E}^H$ are released using "(4.17)".

Example 5

In Fig. 11, the current alias set reference in the heap is initialized to *0*. At time stamp *4*, variables X and Y are aliased. Since the alias set references for variables X, Y are *0*, a new F-bag # 1 is created in the alias set environment, and the bits corresponding to variables X and Y are flagged, and the heap is updated to indicate that variables X and Y share F-bag # 1. Two tuples (*4, 0, 1*) and (*4, 0, 1*) are inserted in $\mathcal{E}^H$. Similar action is repeated when variables Z and W are aliased at time stamp *6*: a new F-bag # 2 is created; $\mathcal{E}^G$ is pdated to indicate that the variables Z and W share the F-bag # 2, and alias history is updated to store two new tuples (*6, 0, 2*) and (*6, 0, 2*). At time stamp *8*, variables X and Z are aliased resulting in aliasing of all four variables X, Y, Z and W. This aliasing results in the F-bag # 3 by deriving D-intersection of F-bags # 1 and # 2; two data parallel search-and-update sequences (one for alias set $\{X, Y\}$ and other for alias set $\{Z, W\}$ are used to update $\mathcal{N}^D$ in $\mathcal{E}^G$ to refer to F-bag # 3; and two new tuples (*8, 1, 3*) and (*8, 2, 3*) are entered in $\mathcal{E}^H$.

Upon backtracking, the field $\mathcal{T}$ is searched to derive the previous F-bags # 1 and # 2 from the field $\mathcal{N}^O$, and new F-bag # 3 from the field $\mathcal{N}^U$. Alignment of $\mathcal{E}^G$ and the $\mathcal{E}^V$ allows aligned update (see Eq. (4.16)) of $\mathcal{E}^G$ to indicate that variables X and Y share F-bag # 1 and variables Z and W share F-bag # 2.

7.1.1. Handling Shared Variables

In a conjunctive goal (or subgoals), a producer may be bound to a D-bag. In such cases, the index of the corresponding D-bag of bindings is stored in $\mathcal{E}^G$ and the D-bag is stored in $\mathcal{E}^D$. If the consumer uses one scalar value at a time, then one value is picked from the D-binding at a time, and the selected binding is released from the D-bag. This process is continued using an iterative loop until the D-bag is $\emptyset$. For a goal having two or more producers, tuple of bindings is derived fro.

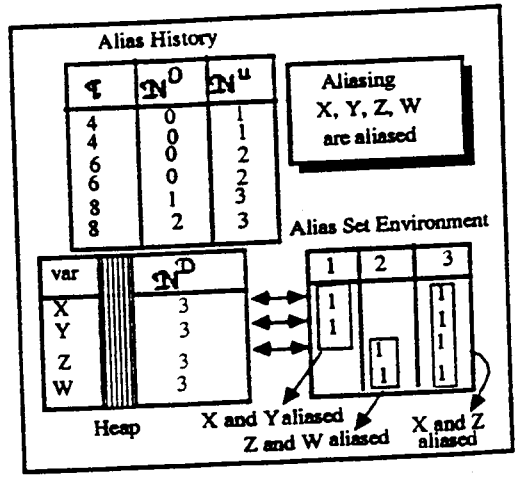


Fig. 11. Aliasing variables associatively.

association of D-bags. The abstract instruction for repetitive selection of values is as follows:

repeat-else-backtrack <*label*>
<*label*>: *next-value* <*binding-vector*>

The first instruction stores the <*label*> into the control stack along with the other information, and goes to the next instruction to pick up a value from the D-bag. Upon backtracking, the control comes back to the instruction "next-value", and the next iteration takes place.

7.2. Handling Multiple Procedures

While deriving the queries for the goals with uninstantiated predicate names, there may be multiple procedures which are unifiable to the same goal, since they share the same object. However, each of these procedure must be solved separately to derive different relations and the related argument values. This is achieved by picking up one clause, identifying the corresponding procedure, and flagging all the unifiable clauses having the same procedure name in a different F-bag. During backtracking, the next clause is picked up in two different steps: if the current procedure has an unprocessed clause, then the next clause of the current procedure is selected. After all the clauses in a procedure are over, a F-bag corresponding to unifiable clauses in next unifiable procedure is picked.

Meta-relations are treated as any other relation with the exception that uninstantiated variables also occur in the place of predicate names.

8. Performance Evaluation

The prototype emulator, which also includes the data parallel SIMD operations, has been implemented using ANSI C. The data demonstrates that the number of operations needed for associative lookup is independent of the number of ground facts. Thirty operations are needed to match a ground fact with two arguments. The number of operations is linearly dependent upon the number of arguments in a query as shown in Table 2. For each extra argument, twelve extra operations are needed to load the value in registers, perform data parallel match, and perform logical ANDing of the previous bit-vector with the new bit-vector obtained during data parallel match.

Table 2. Number of non-shared arguments vs. execution speed.

Number of arguments	1	2	3	4	5	6	7	8	9	10
Parallel operations	18	30	42	54	66	78	90	102	114	126
Scalar operations	2	3	4	5	6	7	8	9	10	11
Total operations	20	33	46	59	72	85	98	111	124	137

For a 20 ns clock supported by current technology, and three clock cycles (load-execute-store cycle), the associative look up speed is eight hundred thousands × number of facts for a set of facts with two arguments. In the presence of data parallel scientific computations intertwined with associative lookup, the peak execution speed is limited by the associative look-up speed which is eighty MCPS (million computations per second) for thousand facts.

The worst case of execution occurs when consumer of a shared variable uses one scalar value at a time. In such cases, the execution speed reduces to two hundred thousand logical inferences per second (LIPS). The slow down is caused primarily due to the overhead of storing the control thread during forward control flow, register set up, and retrieving the control thread during backtracking. Our results show that the overhead of data parallel matching is less than the overhead of storing the control thread during forward control flow which makes the model suitable for handling flat programs with relations having a large number of arguments.

9. Other Related Works

Earlier work on applying search by content for Prolog was described elsewhere [6]. Recently, concurrent work on DAP (Distributed Array Processor) Prolog [8] exploits three paradigms, namely, set based programming, data parallel scientific computation, and associative search to a limited extent. The scheme tries to simulate WAM which causes some limitations since WAM was not designed to exploit associativ

computation. In comparison to our recent work [15], the limitations of this scheme are as follows:

(i) Intertwining of associative search and data parallel computation is negligible, and is not based upon any formal model.
(ii) Left hand side of the set of clauses is not aligned with data parallel environment which causes the overhead of data movement. The concept of data parallel environment is very limited. Only arrays are used for data parallel computation.
(iii) Derivation and reasoning using unspecified relations is not supported.
(iv) Lack of run time allocation in parallel field does not allow indefinite size of sequences.

Associative computation is not exploited for handling aliased variables.

Other schemes [7] simulate pointer based representations using associative memories. The scheme does not exploit the power of association to represent data associatively. The scheme also does not intertwine associative search with data parallel computation. The major differences between the proposed model and WAM based models are as follows:

(i) Unlike WAM, alternative clauses are not traversed sequentially. In our model, clause-body and clause-head are loosely connected through a label. Associative non-deterministic search is used to pick up the next clause. In WAM there is no concept of association of D-bags and associative search on D-bags. At best, WAM makes use of indexing on limited number of arguments.
(ii) Unlike WAM based models, our model is based upon alignment of association D-bags. The alignment of data reduces the data movement during selection of the data elements from program association, and provides tighter integration between logic programming and data parallel scientific computing. In our model, data parallelism is exploited during data retrieval, computation, and data movement.
(iii) Use of data parallel environment in our model allows *vector-vector* $\rightarrow$ *vector* and *vector-scalar* $\rightarrow$ *vector* computations in unit time.
(iv) Unlike WAM based models, our model uses associative computation for aliasing, which reduces the cost of update and access of bindings for aliased variables.

In contrast to the interpretation based conventional data-parallel model proposed in Refs. 9 and 14, this model does not suffer from data sequentiality caused by the presence of multiple-occurrence variables in the goals. The model also handles variable aliasing in the clauses efficiently using the associative data-parallel search and data-parallel assignment property. The major advantage is the synergy of exploiting associative computation and execution of low level code. This work extends our previously reported work [15] by introducing an algebra for associative computation, and reports the benchmark results.

10. Conclusion

In this paper, we described an associative compilation model which integrates high performance intelligent processing, data parallel computing, and associative information retrieval under the framework of logic programming. The model is capable of answering queries with unspecified relations about the given objects. This model benefits from the synergy resulting from associative search, data-parallelism during goal reduction, the use of low level code, data alignment, and data parallel computation to reduce data-transfer and data transformation overhead. Use of persistent global registers reduces the overhead of storing and restoring the environment. The model has been implemented using C^{++} and ANSI C [16, 17]. The benchmarks of the emulator are very encouraging for data intensive computations.

Acknowledgments

I thank Prasad Lokam for compiling the model, Madhavi Ghandikota for implementing the emulator for the abstract instruction sets, and Jerry Potter for some useful discussions related to SIMD computations. I also thank Leon Sterling for proof reading the paper.

References

[1] R. Kowalski, *Logic for Problem Solving* (Elsevier, 1979).
[2] L. Sterling and E. Shapiro, *The Art of Prolog* (MIT Press, 1986).
[3] Y. Kanada and M. Sugaya, "A Vectorization Technique for Prolog without Explosion", *Proceedings of the International Joint Conference of Artificial Intelligence* (1989) pp. 151–156.
[4] A. Takeuchi and K. Furukawa, "Parallel logic programming languages", *Lecture Notes In Computer Science*, vol. 225 (Springer-Verlag, New York, 1986) pp. 242–254.
[5] D. H. D. Warren, "An abstract prolog instruction set", Technical Report 309 (SRI International, 1983).
[6] P. Kacsuk and A. Bale, *The Computer Journal* 30, 393 (1987).
[7] C. D. Stormon, M. R. Brule and J. C. D. F. Riberio, "An architecture based on content addressable memory for the rapid execution of prolog", in *Proc. Fifth International Conference of Logic Programming* (1988) pp. 1448–1473.
[8] P. Kacksuk, "DAP prolog", in *Execution Models of Prolog for Parallel Computers* (MIT Press, 1990).
[9] A. K. Bansal and J. L. Potter, *The International Journal of Engineering Applications of Artificial Intelligence* 5, 247 (1992).
[10] J. L. Potter, *Associative Computing* (Plenum, New York, 1992).
[11] C. C. Foster, *Content Addressable Parallel Processors* (Van Nostrand Reinhold, New York, 1976).
[12] W. D. Hillis and G. L. Steele Jr., in *Communications of the ACM* 29, 1170 (1986).
[13] J. A. Feldman and D. Rovner, *Communications of the ACM* 12, 439 (1969).
[14] A. K. Bansal and J. Potter, "Exploiting data parallelism for efficient execution of logic programs with large knowledge bases", *Proceedings of the Tools for Artificial Intelligence 1990*, (1990) pp. 674–681.

[15] A. K. Bansal, J. L. Potter and L. V. Prasad, "Data parallel compilation and extending query power of large knowledge bases", *Proceedings of the International Conference of Tools for Artificial Intelligence* (1992) pp. 276–283.

[16] M. Ghandikota, "Implementing abstract instruction set for logic programs on associative supercomputers", MS thesis, Department of Mathematics and Computer Science, Kent State University, USA (1993).

[17] L. V. Prasad, "Compiling logic programs to incorporate data-parallelism on associative supercomputers", MS thesis, Department of Mathematics and Computer Science, Kent State University, USA (1993).