Proceedings

Sixth International Conference on

TOOLS WITH ARTIFICIAL INTELLIGENCE

November 6–9, 1994
New Orleans, Louisiana

Sponsored by

IEEE Computer Society

IEEE Computer Society Press
Los Alamitos, California

Washington • Brussels • Tokyo

A Formal Associative Model of Logic Programming and its Abstract Instruction Set

A. K. Bansal, P. V. Lokam, M. N. Ghandikota
Department of Mathematics and Computer Science
Kent State University, Kent, OH 44242 - 0001, USA
E-mail: arvind@mcs.kent.edu

Abstract

Associative computation is characterized by the intertwining of search by content and data parallel computation. This intertwining facilitates the integration of knowledge retrieval and data parallel computation. This paper describes a formal set of architecture independent rules for associative model of logic programming, and an abstract instruction set. The model integrates knowledge retrieval, data parallel computation, and rule based reasoning within logic programming paradigm. An example of abstract instructions has been presented through the compilations of illustrative programs. Benchmark results have been presented. Benchmark result shows that tight integration of rule based reasoning and data parallel computation has reduced overhead on high performance supercomputers

Keywords:
Artificial Intelligence, Associative Computing, Data Parallel Computation, High Performance, Knowledge Retrieval, Logic Programming, Parallelism

[1]

1 Introduction

Associative computation is characterized by seamless intertwining of search by content and data parallel computation [11]. This intertwining facilitates the integration of knowledge retrieval and data parallel computations [1, 2, 3]. This work is an effort to generalize the associative computation model which started with our earlier work to exploit associative computation on SIMD architectures. In this paper, we describe an abstract representation of data, basic set of computation on this abstract data representation, and the corresponding abstract instruction set for a formal model of associative logic programming [3].

The major significance of the associative model is that it supports a large class of queries to derive unspecified relations based on incomplete information about attributes [2], knowledge discovery, reasoning about meta-relations - relations about relations, and queries which integrate search by content, inequality, and data parallel scientific computation. The domain of application for this model are data intensive problems such as geographical information systems, image understanding systems, statistical knowledge bases, and alignment analysis in genome sequencing. For example, in geographical information systems, spatial data structures such as quad-trees and oct-trees can be represented associatively. Different regions having the same values can be identified using associative search on value, and integrated with intelligent rule based reasoning. In large knowledge bases, 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 schemes which need knowledge retrieval, efficient insertion and deletion from a sequence, and efficient manipulation of matrices for heuristic matching and statistical analysis of sequence-alignments can directly benefit from this scheme. Interested readers may refer to [3] for associative representation of abstract data.

The motivation for this abstract representation and computation is

1. to formalize the kernel set of associative computations which form the basis for the integration of knowledge retrieval and data parallel computation, and

2. to facilitate implementation of the associative model of logic programming on different massive parallel architectures.

A compiler and an emulator for the current model have been implemented. The compiler has been written in C^{++}, and the emulator has been written using ANSI C. The emulator supports both scalar and data-parallel computations, and is portable to any architecture which supports a data parallel version of C. The benchmark results show that overhead of shallow backtracking and deep backtracking has been significantly reduced which allows seamless integration of knowledge retrieval, rule based reasoning, and data parallel scientific computation. Associative representation of data reduces the overhead of sequentiality caused by pointer based data representations [11].

2 Preliminaries

We expect the readers to be familiar with logic programming [9]. A logic program consists of rules and facts. Facts are also known as the data base part,

[1] This research was supported in part by equipment grant of Research Council, Kent State University, Kent, OH

and rules are used to infer complex queries. Solving a logical query is based on repeative reduction of a query to subqueries, using rules of the form *Clause-head* :- $Subgoal_1$, ..., $Subgoal_N$ ($N \geq 0$), until the query can be looked up in the database (facts). The heart of goal reduction is unification - a combination of pattern matching and value binding process. At the lowest level, while matching a subquery with the database, the unification process is reduced to lookup.

A *multiple-occurrence variable* occurs more than once in a goal. A *shared variable* occurs in more than one subgoals of a clause. A *producer* is the first occurrence of a shared variable, and generates a bag of values for a shared variable. Other occurrences of the shared variable are *consumers*. A set of variables are *aliased* if all of them share the same value; binding one of the aliased variables automatically binds others with the same value. Aliasing is an equivalence relationship. A *fact* is a rule with empty body. A *simple fact* is a fact with no multiple occurrence variable, and a *complex fact* has at least one multiple occurrence variable.

2.1 Limitations of WAM

Conventional mechanism to implement knowledge bases are based upon WAM [13] which is based upon pairwise unification of a goal with each clause (or fact), and is linearly dependent upon the number of rules (or facts)[2]. The WAM based models are not suitable for data intensive computations due to their linear dependence of data and sequential representation of abstract data. Interested readers may refer to [1, 2] for a discussion of limitations.

2.2 Associative Computation Model

In this subsection, we define abstract data representations, a set of basic computations on these data representations, and briefly describe the computation model for associative logic programming.

A bag is a collection of data items such that there can be multiple occurrences of a value. Two bags are associated if the individual elements in the D-bags are paired using common index. The advantage of pairing by index is that the knowledge of an element in one D-bag is sufficient to derive the value of the corresponding element in the other D-bag. We define two types of data structures, namely, a D-bag and an F-bag and operations on these sets to explain associative computations. Intuitively, D-bags are used to represent data elements, and F-bags are used to select the data elements.

The associative computation model of logic program is based upon mapping the set of clause-heads as an association of D-bags, and mapping the right hand side of each clause by a sequence of low level abstract instructions. The binding environment in the model is represented as an association of D-bags and F-bags. An association of D-bags is used to represent a set of tuples such that each D-bag represents the elements occurring at the same position in the tuples. Searching for a data item in a D-bag derives an F-bag which is used to select a subbag of the D-bag. A goal can be represented as a tuple of data elements such that each element in the goal is searched in different D-bags in the association representing the set of clause-heads. Each search derives a different F-bag. The intersection of these F-bags, when applied to the association, derives the set of unifiable tuples. A data parallel computation performs the same computation on every element of bags (D-bag or F-bag). After the unifiable clauses are identified, the bindings are transferred to the binding environment which consists of D-bags and F-bags. The low level instructions corresponding to the subgoals are executed, and the process is repeated. In the presence of aliasing, the corresponding F-bag is derived by equating the corresponding D-bags.

The generic architecture for an associative computation model can be described as 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. An associative search of a field for a specific value sets up the corresponding mask bits. A vector of mask bits, which corresponds to a F-bag, is used to filter instructions. An instruction is broadcast to each cell simultaneously where it manipulates the data elements, which correspond to D-bags. The flow of control is effected by generating, saving, and restoring the mask bits (or vector of mask bits) based on the results of tests on local data. SIMD architecture satisfies this criteria. However, D-bags and F-bags and computations on them can be implemented efficiently on other massive parallel architectures.

2.3 Abstract Data Representation

A *D-bag*, denoted by $\mathcal{D}$, is defined as an ordered bag which includes null value $\perp$ and top value $\top$. $\perp$ denotes the absence of any value, and $\top$ denotes an undefined value. $\top$ can be instantiated with any value. For example, { *2*, $\perp$, *3* } is a D-bag. However, { *2*, $\perp$, *3* } $\neq$ { $\perp$, *2*, *3* } since D-bags are ordered. The null element $\perp \preceq$ every element in the D-bag.

A D-bag of M-tuples of the form { $< d_{11}$, ..., d_{1M} ..., $< d_{N1}$, ..., $d_M >$ } is stored as an association of M D-bags aligned to each other such that accessing I_{th} element of one bag also gives access to I_{th} element of other D-bags. We denote the aligned D-bags as $\mathcal{D}_1 \bigoplus \mathcal{D}_2 \bigoplus ..., \bigoplus \mathcal{D}_M$. An F-bag is a D-bag with Boolean values *true* and *false*. The Boolean values *true* and *false* are treated synonymously with the values "1" and "0" respectively. We treat *false* (or "0") $\prec$ *true* (or "1"). A F-bag of *1s* is denoted by $\mathcal{F}^1$, and a F-bag of *0s* is denoted by $\mathcal{F}^0$.

We also define the notion of D-inclusion, D-union, and D-intersection of two D-subbags. These notions are different from there set-theoretic counterparts due to the presence of order in D-bags and F-bags, and inclusion of $\perp$ and $\top$ in D-bags, and pairwise comparison of corresponding elements, instead of membership test. A D-bag $\mathcal{D}_1 = \{ d_{11}, ..., d_{1N} \}$ is included in another D-bag $\mathcal{D}_2 = \{d_{21}, ..., d_{2N} \}$ if $\forall I_{(1 \leq I \leq N)}\ d_{1I} \preceq$

[2] WAM model allows limited amount of indexing on the first arguments if all elements are ground

d_{2I}. For the sake of clarity, We refer to the inclusion of a D-bag by D-inclusion, and denote D-inclusion by $\sqsubseteq$. For example, $\{ 4, \perp, 5, 6 \} \sqsubseteq \{ 4, 3, 5, 6 \}$ since $\perp \prec 3$. D-union of two D-subbags $\{d_{11}, ..., d_{1N}\}$ and $\{d_{21}, ..., d_{2N}\}$ derives a new D-bag $\{ d_{31}, ..., d_{3N} \}$ such that $\forall I_{(1 \leq I \leq N)}\ d_{3I} = d_{1I}$ if $d_{2I} \preceq d_{1I}$, $d_{3I} = d_{2I}$ if $d_{1I} \preceq d_{2I}$. We denote D-union by $\bigsqcup$. For example, $\{ \perp, b, c \} \bigsqcup \{ a, b, \perp \}$ derives $\{ a, b, c \}$. D-intersection of two D-subbags $\{d_{11}, ..., d_{1N}\}$ and $\{d_{21}, ..., d_{2N}\}$ 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 $d_{2I} \preceq d_{1I}$. we denote D-intersection by $\bigsqcap$. For example, $\{ 2, 3, \perp \} \bigsqcap \{ \perp, 3, 4 \}$ derives the D-subbag $\{ \perp, 3, \perp \}$.

We denote application of an F-bag on a D-bag to select a D-subbag by $\mathcal{D} \bigotimes \mathcal{F}$. Under the assumption *false* $\prec$ *true*, D-union of F-bags is implemented by logical-OR of the corresponding logical bit-vectors, and D-intersection of F-bags is implemented by logical-AND of the corresponding logical bit-vectors.

3 Rules for Associative Computation

Our model is based upon following sixteen rules of associative computation. There are five types of laws of associative computation: *laws for data association*, *laws for associative search*, *laws for selection*, *laws for data parallel computations* and *laws for data parallel update*.

3.1 Laws of Data Association

This subsection describes three rules of association of data. Rule (1) describes the formation of association, Rule (2) describes the isomorphism of nested associations, and Rule (3) describes the isomorphism under permutation of an association. The implication of isomorphism of association is that the information in an association is not altered by nesting or by permutation.

Rule (1) states that a D-bag of M-tuples is given by the association of M D-bags such that corresponding elements are aligned by index. For example, $\{ a, 2, 3, 4 \} \bigoplus \{ b, 5, 6, 7 \}$ is equivalent to $\{ < a, b >, < 2, 5 >, < 3, 6 >, < 4, 7 > \}$. Rule (2) states that nested associations are isomorphic For example, $\{ 1, 2 \} \bigoplus (\{ 3, 4 \} \bigoplus \{ 5, 6 \})$ is isomorphic to $(\{ 1, 2 \} \bigoplus \{ 3, 4 \}) \bigoplus \{ 5, 6 \}$, and both are isomorphic to $\{ < 1, 3, 5 >, < 2, 4, 6 > \}$. Rule (3) states that permutations in an association are not symmetric but isomorphic: a pair $(x, y) \in \mathcal{D}_1 \bigoplus \mathcal{D}_2$ (such that $x \in \mathcal{D}_1$ and $y \in \mathcal{D}_2$) has a bijective mapping to (y, x) $\in \mathcal{D}_2 \bigoplus \mathcal{D}_1$.

3.2 Laws of Associative Search

This subsection describes two laws of associative search. Fourth rule describes the mapping of a D-bag to an F-bag based upon an associative search of an element, and Rule (5) describes the derivation of information from other D-bags by searching in one of the D-bags in an association. Rule (4) is necessary for unification, and Rule (5) is necessary for deriving the bindings of variables in a goal after unification.

Rule (4) states that associative search of a data element d in a D-bag $\mathcal{D}_1$ (of the form $< d_1, ..., d_N >$ derives an F-bag $\mathcal{F}$ such that if d_j = d then the corresponding element in $\mathcal{F}$ is "1" otherwise "0". For example, associative search of an element 4 in the D-bag $\{ 3, 5, 4, 7, 4, 9 \}$ gives an F-bag $\{0, 0, 1, 0, 1, 0\}$. Rule (5) 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 $\{4, \top, \top \}$ from the tuple $\{ < 4, 5, 6 >, < 3, 7, 9 >, < 4, 9, 10 > \}$ gives an F-bag $\{ 1, 0, 1 \}$ which gives the selected D-subbag as $\{ < 4, 5, 6 >, \perp, < 4, 9, 10 > \}$.

3.3 Laws of Selection

In this subsection, we describe six rules for selection of data elements from an association using an F-bag or computation on F-bags. Use of F-bags and computation on F-bags before selecting D-bags reduces the overhead of data movement overhead. Rule (6) is used to pick up data elements from the corresponding tuples; Rule (7) - Rule (10) are used to reduce the data movement if D-bags of an association are present on different address space; Rule (8) is used to reduces the creation overhead of subbags of a D-bag. Rules (11) and (12) have been implicitly used to derive the property of many low level computations.

Rule (6) 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, $\{ 3, 5, 6 \} \bigotimes \{0, 1, 0\}$ derives $\{ \perp, 5, \perp \}$. Rule (7) states that selecting data elements from two associated D-bags is same as selecting data elements from individual bags and then associating them. For example, $(\{ 4, 5, 6 \} \bigotimes \{ 1, 0, 1 \}) \bigoplus (\{ a, b, c \} \bigotimes \{ 1, 0, 1 \})$ is equivalent to $\{ < 4, a >, < 5, b >, < 6, c > \} \bigotimes \{ 1, 0, 1 \}$ which derives the D-bag $\{ < 4, a >, \perp, < 6, c > \}$. Similarly, $\{(2, 3), (4, 5), (6, 7)\} \bigotimes \{ 1, 1, 0 \}$ is equivalent to $(\{2, 4, 6\} \bigotimes \{ 1, 1, 0 \}) \bigoplus (\{3, 5, 7\} \bigotimes \{ 1, 1, 0 \})$. The computation derives $\{2, 4, \perp\} \bigoplus \{3, 5, \perp \}$ which is equivalent to $\{ < 2, 3>, <4, 5>, \perp\}$. Rule (8) states that data elements of a D-bag selected using an F-bag $\mathcal{F}_1$ includes the data elements selected using another F-bag $\mathcal{F}_2$ if $\mathcal{F}_1 \sqsubseteq \mathcal{F}_2$. For example, $\{ 5, 6, 7 \} \bigotimes \{ 1, 0, 1 \}$ derives the D-bag $\{ 5, \perp, 6 \}$. While $\{ 5, 6, 7 \} \bigotimes \{ 1, 0, 0 \}$ derives the D-bag $\{ 5, \perp, \perp \}$. Rule (9) states that data elements of a D-bag selected by D-union of two different F-bags is same as results of selecting the data elements by applying individual F-bag on the D-bag and then performing D-union on the resulting D-subbags. For example, $\{2, 3, 4\} \bigotimes (\{1, 0, 0\} \bigsqcup \{0, 1, 0 \}) \equiv (\{2, 3, 4\} \bigotimes \{1, 0, 0\}) \bigsqcup (\{2, 3, 4\} \bigotimes \{0, 1, 0 \}) \equiv \{2, 3, 4\} \bigotimes \{1, 1, 0\}$ which derives $\{ 2, 3, \perp \} \equiv \{ 2, \perp, \perp\} \bigsqcup \{ \perp, 3, \perp\}$. Rule (10) states that data elements of a D-bag selected by D-intersection of two different F-bags is same as results of selecting the data elements by applying individual F-bag on the D-bag and then performing D-intersection on the resulting D-subbags of data elements. For example, $\{2, 3, 4\} \bigotimes (\{1, 1, 0\} \bigsqcap \{0, 1, 1 \}) \equiv \{2, 3, 4\} \bigotimes \{0, 1, 0\}$ which derives $\{ \perp, 3, \perp \} \equiv \{\perp, 3, 4\} \bigsqcap \{3, \perp, 4\}$. Rule (11) states that Cartesian product of a bag $\mathcal{D}$ with *true* is equivalent to the set obtained by

associating $\mathcal{F}^1$ with any D-bag, and is equivalent to $\mathcal{D}$ itself. For example, $\{2, 3, 4\} \times \{ true \} \equiv \{2, 3, 4\} \otimes \{1, 1, 1\}$ which results into the D-bag $\{2, 3, 4\}$. Rule (12) states that Cartesian product of a D-bag $\mathcal{D}_1$ with *false* is the set obtained by associating $\mathcal{F}$ with an D-bag, and is equivalent to a null set. A null set ϕ is also represented as a D-bag with every element as $\perp$.

3.4 Laws of Data Parallel Computation

In this subsection, we describe two rules for data parallel computation. Rule (13) states that if any two D-bags are associated and a data parallel computation is performed on the data elements of each bag then the operation is equivalent to performing the same operation on every element of the associated fields. Any computation involving $\perp$ maps onto $\perp$. For example, $\{2, 3, \perp\} *^D \{3, 4, \perp\}$ derives $\{6, 12, \perp\}$. Rule (14) states that if a scalar value *Val* is operated on a D-bag using a data parallel computation, then the data parallel computation is equivalent to taking Cartesian product of the singleton set $\{ Val \}$ with $\mathcal{F}^1$, and performing data parallel computation on the association $(\{ Val \} \times \mathcal{F}^1) \bigoplus \mathcal{D}_1$. For example, $4 * \{2, 3, 4\}$ is equivalent to $\{4, 4, 4\} *^D \{2, 3, 4\}$ which derives the D-bag $\{8, 12, 16\}$.

3.5 Laws of Associative Update

In this subsection, we describes two rules for associative update: Rule (15) describes associative insertion of a tuple in an association, and Rule (16) describes data parallel release of tuples from an association.

Rule (15) states that if a tuple of the form $< d_1, ..., d_N >$ is inserted in an association of bags $\mathcal{D}_1 \bigoplus, ..., \bigoplus \mathcal{D}_N$, then the association is updated to $(\mathcal{D}_1^U \bigoplus, ..., \bigoplus \mathcal{D}_N^U)$ where $\mathcal{D}_I^U$ denotes the updated bag. Each $\mathcal{D}_I^U = \mathcal{D}_I \bigcup \{d_I\}$. Rule (16) states that by associatively searching in one field, the associated data elements in the other field can be released in a unit computation. For example, associative search for a tuple $\{4, \top, \top\}$ from the tuple $\{ < 4, 5, 6 >, < 3, 7, 9 >, 4, 9, 10 > \}$ derives an F-bag $\{ 1, 0, 1 \}$. Complement of $\{ 1, 0, 1 \}$ derives $\{ 0, 1, 0 \}$. Application of $\{ 0, 1, 0 \}$ derives the D-subbag as $\{ < \perp, < 3, 7, 9 >, \perp > \}$ deleting all the tuples which have value 4.

4 An Associative Abstract Machine

The compilation model maps the program as a pair of associations of the form $< \mathcal{L} \bigoplus \mathcal{P} \bigoplus \mathcal{A}_1 \bigoplus, ..., \bigoplus \mathcal{A}_N, \mathcal{L} \bigoplus \mathcal{C} >$. The first element of the pair represents D-bag of clause-head tuples, and second element of the pair represents the clause-body tuples. $\mathcal{L}$ is the D-bag of labels connecting clause-heads to sequence of compiled abstract instructions of the corresponding clause-body, $\mathcal{P}$ is the D-bag of procedure-names, $\mathcal{A}_I$ is the D-bag of I_{th} argument in set of the clause-heads in a program, $\mathcal{C}$ is the D-bag of a sequence of compiled instructions corresponding to set of clause-bodies in the program such that each element $c_i \in \mathcal{C}$ is a sequence of instructions corresponding to one clause-body.

The computation model has following components:

1. an associative data parallel abstract instruction set,
2. an associative representation of clause-heads,
3. an associative heap to store the bindings of the output variables, shared variables, and scalar bindings.
4. a data parallel binding environment which are aligned to the association of parallel field representing clause-heads, and to the processing elements. The data parallel environment is used to store the F-bags derived during compile-time analysis, F-bags to mark unifiable clauses, and D-bags to store bindings for a variable. Data parallel binding environment is a stack of D-bags and F-bags.
5. an associative scheme to handle the aliased variables,
6. an associative control stack to store a control-thread,
7. a sequence of global registers - an associative equivalent of scalar registers in WAM to store the bindings of arguments of a subgoal.

Due to space limitation, we are omitting graphical representation of the model. The implementation details of the computation model is available in [2,3]. The model exploits run time execution efficiency both at the data level during data parallel goal reduction by treating the clause-heads as data 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 potential bindings for goal variables. Pre-call processing has four components, namely, *matching constant arguments in a goal to corresponding D-bags $\mathcal{A}_I$, data parallel equality of corresponding D-bags $\mathcal{A}_I$ and $\mathcal{A}_J$ to handle multiple-occurrence variables, and D-intersection of F-bags, derived by matching each goal arguments with the corresponding D-bag of clause arguments, to derive F-bag of unifiable clauses* [2, 3]. Pre-clause processing is used to test the presence of unifiable clauses using the F-bag of unifiable clauses, and passing control to the right clause nondeterministically using shared label. 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. Every time a new goal is invoked the frame-pointer for control stack and data parallel binding environment are updated; old frame-pointers are saved in the control stack; and the D-bags and F-bags are addressed as an offset with respect to the frame pointer. If at any time the F-bag representing unifiable clauses is empty then backtracking occurs, information is retrived from the control stack, and the previous environment is restored.

There are four rules for associative goal reduction. The first rule is the *rule of matching constants.* The

rule states if goal argument is a constant then matching the goal argument Arg_I^G with the corresponding field $\mathcal{A}_I$ derives an F-bag $\mathcal{F}_I$. The rule is used to prune those clauses which do not match for a particular argument. The second rule is the *rule of data parallel equality-test.* The rule states that if two goal arguments Arg_I^G and Arg_J^G have occurrence of same variable then F-bag is derived by performing equality test on both the D-bags $\mathcal{A}_I$ and $\mathcal{A}_J$. The third rule is the *conjunctive rule of non-unifiable clauses* which takes the D-intersection of F-bags derived by matching individual goal arguments with corresponding D-bags in the program. The fourth rule is rule of unification, and is generally used for handling aliased variables.

A detailed scheme for the associative computation model is given in [2, 3].

5 An Abstract Instruction Set

The Abstract instruction set of the Dprolog compiler has been divided into five classes, namely, *pattern matching instructions for goal reduction, data selection and data movement instructions, control instructions,* and *data-parallel computation.* Pattern matching instructions are mainly used during data parallel goal reduction, handling aliasing, and unification. Data movement instructions are used to transfer data between registers, transfer data between between D-bags in the data parallel binding environment, and between heap and registers. Control instructions test the F-bag of unifiable clauses. If the F-bag is empty then control backtracks to select another binding for a producer. Logical parallel instructions are used to select the unifiable clauses by deriving the D-intersection of the corresponding F-bags obtained by individual matches, finding the D-union of aliased sets, and handling negation by complementing the F-bag. Arithmetic computation instructions are of three types, namely, *scalar-scalar* $\rightarrow$ *scalar*, *vector-vector* $\rightarrow$ *vector*, and *scalar-vector* $\rightarrow$ *vector.*

During the discussion of abstract instruction set, we will denote I_{th} argument of a goal by Arg_I^G and the corresponding D-bag in the program by $\mathcal{A}_I$. Arg_I denotes both Arg_I^G and $\mathcal{A}_I$, U denotes F-bag of unifiable clauses, F denotes F-bag of simple facts without aliased variables, B denotes F-bag of complex clauses (facts with aliased variables or clauses with non-empty clause-body), C indicates unifiable complex clauses, and T_I denotes a F-bag to store temporary result.

5.1 Goal Reduction

Four abstract instructions are used during goal reduction: *Match_Register_Arg*, *Test_And_Backtrack*, *And_Bit_Vectors*, *Store_Vector_Id*, *DP_unify*. The instruction *Match_Register_Arg* matches a goal argument Arg_I^G with the corresponding D-bag representing a procedure argument $\mathcal{A}_I$ and finds out resulting F-bag of unifiable clauses. The instruction *Test_And_Backtrack* forces the control to backtrack if the F-bag of unifiable clauses is empty. The instruction *And_Bit_Vectors* derives D-intersection of two F-bags to prune non-unifiable clauses. The resulting F-bag selects the unifiable clauses. The instruction *Store_Vector_Id* stores the the reference of the bindings, which is stored as a D-bag in the data parallel binding environment, of a goal-variable Arg_I^G. The instruction *DP_unify* is used to perform data parallel equality test between two D-bags representing two program arguments for multiple-occurrence goal variables. The goal reduction process iteratively calls the sequence of the instructions *Match_Register_Arg*, *And_Bit_Vectors*, and *Test_And_Backtrack* until all the goal-arguments are processed or the F-bag of unifiable clauses is empty.

5.2 Pre-clause Processing

Four abstract instructions are used during pre-clause processing: *Compl_Bit_Vector*, *And_Bit_Vectors*, *Test_And_Return*, and *Try_Me_Else*. The instruction *Compliment_Bit_Vector* derives the compliment of every element of an F-bag into a new F-bag. The instruction *Test_And_Return* is used to pick up randomly another simple fact corresponding to *true* entry in the F-bag of unifiable clauses. if the F-bag of unifiable clauses is empty, the control is passed on to code-area of complex facts or clauses with non-empty body. The instruction *Try_Me_Else* is similar to traditional WAM instruction, and is used to select alternate complex clauses (complex facts or clauses with non-empty body). The instruction tests the F-bag of unifiable-clauses, and picks up the next complex clause randomly, looks up the corresponding label in the data area, and passes the control to the code area sharing the same label.

5.3 Processing Individual Clauses

There are multiple types of abstract instructions which are used in processing an individual clause. Some common instructions are *unify*, *Copy_Logical_Register*, *Call*, *Return*, *Continue*, *Load_Register*, *Repeat_Else_Backtrack*, and *Load_Next_Value*. The instruction *unify* is used to unify two logical terms, and is also used for handling variable aliasing in the clause-head. The instruction *Copy_Logical_Register* is used to move the data from one register to another. However, instead of actually moving the data, the reference to data is moved. The advantage of copying the reference is that attributes related to a data element are not physically moved reducing the run time overhead. The instruction *Call* is used to invoke the procedure corresponding to the subgoal. The instruction *Return* is used to return the control to the calling procedure. The instruction *Load_Register* is used to load a physical value (or reference of a data parallel binding) from the heap into a register.

5.4 Producer-consumer Relationship

Handling producer consumer relationship is divided in two parts: a producer produces a D-bag and the corresponding consumer is in data parallel-goal, or producer produces a D-bag and consumer is in a goal which consumes one scalar value at a time. The example for the first case is a user-defined goal followed by an arithmetic operation, a set operation, or a comparison operation. The first case is handled by data parallel computation on the consumer variable. The second case includes at least

two subgoals such that goal with the corresponding consumer occurrence uses associative search in the clause-heads, or generates multiple values for at least one consumed value. The second possibility is handled by repeative backtracking to fetch a new value and processing the value until the D-bag (bound to the producer) is empty. This repeative backtracking is achieved by a pair of abstract instructions *Repeat_Else_Backtrack* and *Load_Next_Value*: the instruction *Repeat_Else_Backtrack* keeps reverting the direction of control flow (from backtracking to forward control flow) and the instruction *Load_Next_Value* picks up next scalar value from the given vector.

In addition ot these instructions, there are multiple data parallel arithmetic operations, data parallel arithmetic comparison operations, data parallel logical operations.

6 An Illustrative Example

In this section, we explain the most commonly used abstract instructions through a series of compiled programs. Example 1 illustrates the compilation of programs with simple facts, complex facts with aliased variables, and shared variables between subgoals.

Example 1:

The program has three procedures, namely, *p/2, q/2, r/2*. The procedure *p/2* illustrates the intertwining of simple ground facts and complex facts caused by presence of aliasing. The procedure *q/2* is a D-bag of simple facts. The procedure *r/2* is a mixture of ground fact and a non-unit clause. The right hand side of non-unit clause *r/2* exhibits producer-consumer relationship.

p(1,2). p(2, 3). p(3, 4).
p(X, X). * Complex fact due to aliasing *
q(2, 1). q(3, 2). q(4, 3). q(5, 4). * Simple facts *
r(2, 2).
r(X, Y) :- p(X, Y), q(2, Y). * Complex clause *

The Compiled Code: We give the compiled code for the above program, and the corresponding operational semantics of the abstract instructions.

% Pre-call processing instructions for procedure *p/2.*

DP_p/2:	*Match_Register_Arg*	Arg_1, U_0
	Test_And_Backtrack	U_0

This instruction restores the previous environment if the F-bag selecting the set of unifiable clauses $U_0 = \phi$ and backtracks.

	Match_Register_Arg	Arg_1, U_0
	Test_And_Backtrack	U_0
	And_Bit_Vectors	$F_0, \quad U_0, \quad B_0$

This instructions derives the D-intersection of two F-bags F_0 and U_0 and stores the result in a new F-bag B_0.

	Store_Vector_Id	$B_0, \quad Arg_1$

This instruction stores the index of F-flag B_0 and the index of D-bag $\mathcal{A}_I$ into the heap as a reference to bindings for the variable Arg_I^G.

	Store_Vector_Id	$B_0, \quad Arg_2$

% Pre-clause instructions for procedure *p/2*

	Compl_Bit_Vector	$F_0, \quad T_1$
	And_Bit_Vectors	$T_1, \quad U_0, \quad C_0$
	Test_And_Return	$B_0, \quad L_{10}$

This instruction picks up another simple fact if F-flag B_0 is not empty otherwise it passes control to label L_{10}.

L_{10}:	*Try_Me_Else*	C_0

This instruction restores the previous environment if $C_0 = \phi$ otherwise it picks up next complex clause, and passes the control to the corresponding label.

% Clause Processing for procedure *p/2*

	CF3_p/2:*Unify*	Arg_1, Arg_2

This instruction is used to alias two arguments having same variable name.

	Return	

This instruction returns the control to calling Procedure.

% Instructions for procedure *q/2*

DP_q/2:	*Match_Register_Arg*	Arg_2, U_0
	Test_And_Backtrack	U_0
	Match_Register_Arg	Arg_2, U_0
	Test_And_Backtrack	U_0
	And_Bit_Vectors	$F_0, \quad U_0, \quad B_0$
	Store_Vector_Id	$B_0, \quad Arg_1$
	Store_Vector_Id	$B_0, \quad Arg_2$
	Compl_Bit_Vector	$F_0, \quad T_1$
	And_Bit_Vectors	$T_1, \quad U_0, \quad C_0$
	Test_And_Return	$B_0, \quad L_{23}$
L_{23}:	*Try_Me_Else*	C_0

% Instructions for procedure *r/2*

DP_r/2:	*Match_Register_Arg*	Arg_1, U_0
	Test_And_Backtrack	U_0
	Match_Register_Arg	Arg_2, U_0
	Test_And_Backtrack	U_0
	And_Bit_Vectors	$F_0, \quad U_0, \quad B_0$
	Store_Vector_Id	$B_0, \quad Arg_1$
	Store_Vector_Id	$B_0, \quad Arg_2$
	Compl_Bit_Vector	$F_0, \quad T_1$
	And_Bit_Vectors	$T_1, \quad U_0, \quad C_0$
	Test_And_Return	L_{34}
	B_0,	
L_{34}:	*Try_Me_Else*	C_0

P_2_R1:	*Copy_Logical_Register*	Arg_1, Arg_1

This instruction copies the reference to the actual value from Arg_1 of the clause-head to the register representing Arg_1 of the first subgoal. The advantage of storing reference is that data movement is avoided.

	Copy_Logical_Register	Arg_2, Arg_2
	Call	*DP_p/2*
	Continue	L_{39}
L_{39}:	*Load_Register*	$Arg_1, T_1, \quad 2$

This instruction loads value *2* in a physical register T_1 and sets up a reference in another register representing

Arg_1.

Copy_Logical_Register Arg_2, Arg_2

Repeat_Else_Backtrack L_{42}

This instruction stores the label L_{42} into the control stack, and passes control to the next instruction. Upon backtracking, if the F-flag of unifiable clauses is empty then backtracking takes place. This instruction simulates failure driven iteration until all the unifiable clauses have been tested.

L_{42}: *Load_Next_Value* Arg_2

This instruction gets the index of the D-bag for the variable in Arg_2 from the heap, and retrieves the next value from the D-bag.

Call *DP_q/2*

Return

6.1 Performance Evaluation

The prototype emulator has been implemented using ANSI C. The emulator is portable to any architecture which supports data parallel version of C. The results demonstrate that the number of operations needed for associative lookup is independent of 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. For each extra argument, nine 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.

For a 10 ns clock supported by current technology, and three clock cycles (load-execute-store cycle), the associative look up speed is 1.2 million × number of facts for a set of facts with two arguments. In the presence of data parallel scientific computations intertwined with associative lookup, the peek execution speed is limited by the associative look-up speed which will be one hundred and twenty MCPS (million computations per second) for thousand facts.

When two subgoals of a rule share variables, and the second subgoal is not a data parallel computation, the data elements in vector bindings for shared variables are processed one at a time. This scenario is the worst case for execution, and the execution speed reduces to four hundred thousand logical inferences per second (LIPS) for one shared variable. 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.

7 Conclusion and Future Work

In this paper, we describe a formal model of associative logic programming and an abstract instruction set. The abstract instruction set integrates associative knowledge retrieval and data intensive computations. The abstract instructions are portable to any SIMD machine and with some modifications to other distributed memory massive parallel architectures. Currently we are extending the model to execute in an architecture independent way in distributed memory massive parallel MIMD architectures.

References

[1] A. K. Bansal and J. L. Potter, "An Associative Model to Minimize Matching and Backtracking Overhead in Logic Programs with Large Knowledge Bases", *The International Journal of Engineering Applications of Artificial Intelligence*, Permagon Press, Volume 5, Number 3, (1992), pp. 247 - 262.

[2] A. K. Bansal, J. L., Potter, and L. V. Prasad, " Data Parallel Compilation and Extending Query Power of Large Knowledge Bases," In the Proceedings of the International Conference of *Tools for Artificial Intelligence 1992*, pp. 276 - 283.

[3] A. Bansal, "An Associative Compilation Model for Tight Integration of High Performance Knowledge Retrieval and Computing", International Journal on Artificial Intelligence Tools, World Scientific Publishers, in press.

[4] J. A. Feldman and D. Rovner, "An Algol Based Associative Language," *Communications of the ACM*, Volume 12, No. 8, August 1969, pp. 439 - 449.

[5] D. Gries, "The Science of Programming", Monograph, Springer Verlag, Newyork, 1987.

[6] K. Hwang and F. A. Briggs, *Computer Architecture and Parallel Processing*, Mcgraw Hill Book Company, New york, USA, (1984).

[7] P. Kacsuk, and A. Bale, "DAP Prolog: A Set Oriented Approach to Prolog," *The Computer Journal, Vol. 30, No. 5, 1987*, pp. 393-403.

[8] K. Knobe, J. D. Lukas, G. L. Steele, "Massively Data Parallel Optimization", The 2nd Symposium of Massively Parallel Computation, Fairfax, Virginia, 1988, pp. 551 - 558.

[9] Kowalski, R., *Logic for Problem Solving*, Elsevier-North Holland, (1979).

[10] Z. Manna and R. Waldinger, "The Logical Basis for Computer Programming", Volume1: Deductive Reasoning, Addison Wesley, 1985.

[11] J. L. Potter, *Associative Computing*, Plenum Publishers, Newyork, (1992).

[12] A. Takeuchi and K. Furukawa, "Parallel Logic Programming Languages", *Lecture Notes In Computer Science, Vol. 225*, Springer Verlag, Newyork, (July 1986), pp. 242 - 254.

[13] D. H. D. Warren, "An Abstract Prolog Instruction Set", *Technical Report 309*, SRI International, (October 1983).