Abstract

This paper describes the compilation of an associative model of logic programs extended to incorporate vectors and associations to integrate data-parallelism present in associative supercomputers and efficiency present in low level code. The integration is based upon treating the clause-heads as a two dimensional associative table, and compiling subgoals to low level instructions. This hybrid model uses associative search by content to extend the query power of logic programs by deriving the unspecified relations among the given objects; and retains the advantages of data-parallel goal reduction, low overhead of backtracking due to associative search, and low data transfer overhead due to use of global registers.

Keywords: Artificial intelligence, Associative computing, Compilation, Data-parallelism, Knowledge base, Logic programs.

1 Introduction

In last ten years, the logic programming paradigm has become a popular tool for AI systems and knowledge representation. However, the conventional models answer a limited class of queries of the form \(<relation-name>(<object1>, <object2>)\) where \(<object1>\) or \(<object2>\) can be variables but \(<relation-name>\) must be a constant. For example, conventional models answer the query \(parent(tom, X)\)\. However, conventional models cannot answer a large class of queries, which humans are capable of answering, needed in real world knowledge retrieval and processing. A few of the classes of queries are

1. queries which relate one or more objects without apriori knowledge of the relationship,
2. queries which reason about meta-relations - relations relating relations - about the objects, and
3. generic queries with incomplete information.

In Example 1, conventional models answer queries such as “Who is a sister of Tom?”; “Who is a brother of Meera?”; “Who is a parent of Meera?”; “Who is a parent of John” etc. These models can not answer queries such as “How are Tom and Meera related?”; “How are Meera and John related?”; “Specify the relatives and their corresponding relationships to Tom?”; and “Who are the blood-relatives of Tom?”\. The corresponding query formulations are given in Table I: the first column illustrates the class of queries, the second column illustrates formulation, the third column shows reduction scheme used, and the fourth column shows the processing capability of conventional systems. For representational convenience, we abbreviate the query types “goal reduction” by \(G\), “fact lookup” by \(F\), and “meta-relations” by \(M\). The variables \(P\) and \(X\) are uninstantiated. In Table I, the class of queries with uninstantiated variables in place of predicate-names can not be answered by conventional systems.

Example 1:

\[
\begin{align*}
\text{blood-rel} & (\text{sister}). \\
\text{blood-rel} & (\text{brother}). \\
\text{blood-rel} & (\text{parent}). \\
\text{brother}(X, Y) & : - \text{parent}(Z, X), \text{parent}(Z, Y), \\
& \quad \text{male}(X), \text{not-same}(X, Y). \\
\text{sister}(X, Y) & : - \text{parent}(Z, X), \text{parent}(Z, Y), \\
& \quad \text{female}(X), \text{not-same}(X, Y). \\
\text{parent}(\text{tom, meera}). & \quad \text{parent}(\text{linda, meera}). \\
\text{parent}(\text{tom, john}). & \quad \text{parent}(\text{linda, john}). \\
\text{male} & (\text{tom}). \quad \text{male} (\text{john}). \\
\text{female}(\text{meera}) & \quad \text{female}(\text{linda}).
\end{align*}
\]

In this paper, we describe an associative model of logic programs for compiling large knowledge bases on associative SIMD supercomputers. The approach overcomes two limitations of conventional logic programming, namely, the overhead in handling of data intensive knowledge bases, and the inability of inferring about unspecified relations using the information about objects. The model is based upon:
Table 1: Limitations of conventional systems

1. the representation of clause-heads in a two dimensional associative table to use the physical alignment of the arguments;
2. unit-step data parallel pattern matching available on associative SIMDs which makes goal reduction and searching almost independent of the knowledge base size [2, 3]. The basic idea is to use data parallel pattern matching to match a goal term with the corresponding terms in the clause heads. For example, in Figure 1, only 3 data parallel matches are needed for data parallel reduction of the goal $P(H, k)$;
3. releasing the bindings in a unit step during backtracking. For example, in Figure 2, the current value of time stamp (20) is searched associatively in one data parallel operation, and all the bindings are associatively released using one operation; and
4. the representation of subgoals as low level procedural code analogous to WAM [15], and the use of global data registers to achieve a reduction of the data transfer overhead.

The current model has the following advantages over the previous interpretation based on a pure data parallel model [2, 3]:
1. Previous model represented the full program as data, and suffered from subgoal copying costs during new resolvent formation after the goal reduction.
2. The current model uses associative constructs to avoid the overhead of dereferencing aliased variables - uninstantiated variables treated as same variables due to unification.
3. The representation of each clause in a single row of the two dimensional associative table facilitates the data-parallel pruning of the clauses even in the presence of multiple occurrence goal variables - variables occurring more than once in a goal predicate.
4. The query power of logic programming has been increased to answer queries with unspecified relations for the given objects.

Section 2 briefly describes associative computing, WAM - the compilation model on conventional machines, advantages of exploiting data-parallelism in logic programs, and some new definitions. Section 3 describes the architecture of the associative compilation model. Section 4 describes the associative features of the current model to handle data parallel clause pruning in the presence of multiple occurrence goal variables, handling aliased variables, and deriving bindings for relations. Section 5 describes the model behavior, and the last section concludes the work.

2 Background

We assume familiarity with logic programming [8, 13] and the conventional implementations [15]. We also assume the reader to be somewhat familiar with the concepts of associative processors [4, 10], associative computing, and data parallelism in logic programs [2, 3].

2.1 Associative computing

An associative computer consists of an array of processing cells which are operated in parallel by a single instruction. This type of parallelism is known as data...
parallelism. Each cell has memory, a processing element, and its own mask bit which is set selectively to filter instructions. All the instructions for a program are broadcast to all cells; the flow of control is effected by generating, saving, and restoring the mask bits based on the results of tests on local data [4, 11].

An associative search of a field for a specific value sets up a mask field which is stored and manipulated using data parallel operations. Data items of the same data type are stored in a parallel field - a two dimensional memory organization of columns and rows with a cell processor attached to each row. Each row stores a data item. The data items are stored in an unordered fashion in physically non-contiguous rows. An association is defined recursively as a tuple of fields or associations such that given an index, all of the data items in the different fields associated with the index can be accessed using one associative search. On associative computers, data parallel operations are independent of the data set size since each data element can be mapped on a single processing element [11], and the same instruction is broadcast to all of the processing elements in a unit 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 $O(\text{number of elements/number of processing elements})$. An operation is data sequential if the same operation is performed individually on the elements, one element at a time, of a field in some order.

2.2 Definitions in logic programming

Variables begin with capital letters, and constants begin with small letters. A multiple occurrence variable occurs more than once in a predicate. A single occurrence variable occurs once in a predicate. Two uninstantiated variables are aliased if they unify with each other. Aliasing of two variables belonging to two different sets forms a new set, derived by taking union of two sets, such that all of the variables are aliased to each other. Meta relations treat relations as objects. In Example 1, the predicate blood_rel/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 “$P(\text{tom}, X)$” and “$P(X, \text{tom})$” respectively. The uninstantiated variables $P$ and $X$ indicate unspecified relation-names and unspecified object respectively.

2.3 WAM - the conventional model

Briefly, a WAM [15] 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 Figure 4). A heap is a global 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 local stack) of the procedure being executed. The use of argument registers reduces the data transfer time significantly during a procedure call, and has the same application as the use of registers in assemblers.

2.4 Exploiting data-parallelism

Logic programs are well suited for exploiting data parallelism [2, 3] on associative supercomputers by

1. matching a goal position with the corresponding argument in all the clauses in one data parallel operation. An association of parallel fields is used to hold all the elements of an argument in the different clauses. This process matches the ground values to identify non-unifiable clause-heads, and derives the potential bindings for the goal variables. The data parallel match reduces the shallow backtracking significantly [2, 3];

2. associatively releasing the required bindings in the heap in a single operation during backtracking. A data parallel associative search for the current time stamp, used to mark the instantiation of variables in the heap, is performed in a single operation [2, 3] (see
Figure 2). This process significantly reduces the overhead of garbage collection during backtracking;

(3) using associative tables instead of conventional stacks during unification of two terms [2, 3]; and

(4) using unit data parallel operations on all the elements of the same sequence. In conventional logic programs, such operations are dependent on the size of the input data-set due to use of tail recursion, and are at best close to linear iteration time.

In the paper, we will denote the bit-vector containing the information about unifiable clauses as clause-filter-vector, and flag the unifiable clauses by $1$, and non-unifiable clauses by $0$.

3 The architecture

The compilation model maps the clause heads to data for efficient pattern matching during goal reduction, and maps the subgoals to low level code. The data representation of the clause heads uses an association of $N + 2$ ($N$ is number of arguments in the clause) physically aligned parallel fields: the first field stores the label marking the first low level instruction in code area representing subgoals of the corresponding clause, the second parallel field stores the predicate-names, and the remaining $N$ parallel fields store the arguments. The advantages of physical alignment of arguments and sharing of same labels are described in Section 4.1 and Section 5 respectively. The model has the following components:

1. a data parallel abstract instruction set,

2. a data parallel representation of clause heads including facts. The clause head representation also contains the name of the predicate to answer unspecifed relation-name related queries,

3. compile time vectors associated with the data parallel representation of clause heads, to mark the non-unifiable clauses obtained at compile time,

4. flag vectors to mark non-unifiable clauses and goal variable bindings derived at run time,

5. a heap to store the bindings of the output variables and shared variables,

6. an alias vector array associated with heap to store the aliasing information,

7. an alias management table to store the information needed to restore the previous aliasing information during backtrack,

8. a control stack to store control-thread, and

9. an environment vector - an equivalent of scalar registers in WAM to store the bindings of goal arguments.

Figure 5 describes the model. The abstract instruction set consists of instructions for data parallel matching of the goal arguments to the corresponding clause heads, binding a goal variable to a vector of values, extracting a scalar value from a vector, logical operations on bit vectors, testing and backtracking for a null bit-vector, shallow backtracking, data movement between the heap and the global registers, and procedure calls. However, the details of individual instructions [12] are omitted due to page limitation.

The environment vector consists of two data structures, namely, the global registers set and an array of indices. In contrast to scalar registers in WAM, the global registers set is an associative array of values. However, the concept of registers has the same purpose - to avoid memory access for variable lookup. Unlike, the limited number of general purpose scalar registers in conventional machines, the number of global registers is limited by the number of processing elements in the computer which is in the order of $4K$ to $64K$. Each element of the array of indices corresponds to the state of the registers during a goal reduction. The register values in a subgoal are set up by storing the index of the global registers in the corresponding argument slot.

The heap is an association of the variable-id, binding type (scalar or vector), element type, and scalar binding (or clause-filter-vector index in the data parallel bindings). In addition the heap is associated with the alias-vector which associatively handles the bindings. A variable can be instantiated to a scalar value, or a vector of values during the data parallel binding of the goal variable. If the variable is instantiated to a vector then the binding type is set to a vector, and the index of the corresponding binding vector in the data parallel binding is stored in the heap. When the next value is extracted from the vector, it is stored as
a scalar value in the scalar binding slot of the heap and in the environment vector for further lookup.

The control stack is an association which stores the time-stamp of the previous subgoal to be reduced, the label for the code area during backtrack, the index of the partitions in the global registers used for a subgoal, the index for the base of the compile time filter vectors used in the subgoal, the index for the base of the alias filters used in the subgoal, and the index for the base of the data-parallel binding vectors derived at run time in the subgoal. During backtracking, the time-stamp is decremented, the corresponding links are identified, and the corresponding environment is restored. The control jumps to the code area given by the restored label.

The set of aliased variables are handled associative using two data structures namely, alias vector array and alias management table: an alias vector array is used to hold the different sets of aliased variables, and an alias management table is used to retrieve the information about previous sets during backtracking. The alias vector array is associated with a heap to achieve a physical alignment with the heap variables. Each element of the alias vector array is a bit vector flagging the variables in the set of aliased variables. A detailed description of the aliasing mechanism is given in Subsection 4.1.

4 Efficiency and query power

Associative compilation is based upon representing the clause-heads as an association of a predicate and its $N$ arguments such that the arguments in the same clause are physically aligned. This alignment facilitates

1. the representation of two argument positions of the set of clauses as two physically aligned sets. The logical bit vector flagging the subsets of data in these two sets can be logically ANDed or logically ORed to derive intersection or union of the subsets in unit time (see Figure 6) [6].

2. the identification of the predicate name when all the given arguments of a clause match.

Example 2:

In Figure 7, the set \{linda, john, meera, tom, arvind, jerry\} is represented as a parallel field. Logical bit vectors $L_1$ and $L_2$ represent the subsets \{linda, tom\} and \{linda, meera, jerry\} respectively. The ORing of the logical bit vectors $L_1$ and $L_2$ yields logical bit vector $L_3$ which gives the union \{linda, tom, meera, jerry\}. Similarly, logical bit vectors $L_4$ and $L_5$ represent subsets \{linda, john, arvind, jerry\} and \{linda, meera, tom, arvind\} respectively. The logical ANDing of the logical bit vectors $L_4$ and $L_5$ gives $L_6$ which represents the intersection \{linda, arvind\}.

Figure 6: Subset union and intersection

The advantage of physical alignment is that various arguments of all the clauses can be pairwise compared in a data parallel manner. Given a multiple occurrence goal variable occurring in two arguments, the corresponding parallel fields are compared using one data-parallel equality test to prune all the clauses with unequal values in the two argument positions (see Figure 7). This gives an advantage over the previous scheme of representing the predicate as tree and mapping trees as vectors [2] which caused physical disalignment of arguments. In Figure 8, data parallel equality test prunes the first clause since the argument values $a$ and $b$ are not equal.

Figure 7: Multiple occurrence goal-variables

4.1 Handling aliasing effectively

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. The associative programming paradigm can handle aliasing very effectively. The process involves

1. aligning a alias-vector - a bit vector -, with the heap association,

2. marking positions in the alias-vectors corresponding to the union of two sets of aliased variables. This is done by ORing the logical bit-vectors representing two previous alias sets; and

3. creating an alias management table which is an association of three vectors, namely, time-stamp, old-filter-index, and new-filter-index.

Aliasing of uninstantiated variable(s) generates a new aliasing vector. Instantiating any of the vectors,
automatically instantiates all the aliased variables by using an associate search and data-parallel substitution in unit time. Accessing the binding for any aliased variable needs a unit time associative search under this scheme. During backtracking, previous alias sets are recovered by identifying the old-filter indices using associative search for the current time stamp in alias management table, and updating the alias-vector-reference in the heap in a data-parallel manner (see Figure 8).

Example 3:
In Figure 8, the alias vector reference in the heap is initialized to 0. At time stamp 4, variables X and Y are aliased. Since the alias vector references for variables X, Y are 0, a new vector (vector # 1) is created in the alias vector array, and the bits corresponding to variables X and Y are flagged; the corresponding entries in the alias-vector-reference for variables X and Y are updated to I in the heap; and the alias management table stores the information about the old vector information to restore the values during backtracking. Similar action is repeated when variables Z and W are aliased at time stamp 6: vector # 2 is created, and the corresponding entries in alias-vector-reference for variables Z and W are updated to 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 is achieved by ORing logical vectors # 1 and # 2, and storing the result in a new vector (# 3). In two data parallel operations - one for each subset -, the corresponding variables in the previous subsets are identified, and the corresponding entries in alias-vector-reference are updated to 3. Upon backtracking, before decrementing the time stamp; corresponding old indices I and 2 are identified in the "old index field" in the alias management table using data parallel associative search for the value 8 in the "time stamp field"; the old subsets { X, Y} and { Z, W} are identified using the association of bit vectors in alias vectors I and 2 with the corresponding heap variables in a data parallel manner; the alias vector reference is updated with the old value 2 for variables Z and W in one data parallel assignment, and updated with index value I for the variables X and Y in second data parallel assignment.

4.2 Relations for objects
Given a query of the form \( F(X, Y) \), any of the three variables may be uninstantiated. Since all three variables are physically aligned in the two dimensional associative table representation of the clause-heads, any combination of seven queries \(^3\) can be answered in contrast to four queries answerable by conventional models. The query about unspecified relation-names is similar to 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 have to be handled in contrast to the case when the predicate names are instantiated (see Figure 1 and Figure 9).

5 The model behavior
The model behaves at the data level during data parallel goal reduction by treating clause-heads as data for efficient pattern-matching, and behaves at the control level during the execution of the code of the corresponding subgoals in the selected clause. The clause-head in the data area and the first instruction in the code area representing the subgoals of the corresponding clause are linked through the same label. After the data parallel goal reduction, the corresponding clause-label in the data area is picked, and control jumps to the corresponding instructions sharing the same label.

Example 4
In Figure 9, after the data parallel goal reduction of a goal with an uninstantiated variable in the place of the predicate-name, three clauses with labels 1$ (in the procedure with predicate name p1), 4$ and 6$ (both 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 label 6$. Upon the failure of the code with label 6$, the control jumps to the

---

\(^3\) the query with all three variables is meaningless.
5.2 Pre-clause processing

There are two types of instructions which are executed prior to clause processing:

1. Instructions which derive the vector of potential bindings for a goal variable, and
2. Instructions which jump to the codes associated with filtered unifiable clauses.

The run-time jump to the specific code of a unifiable clause is one of the major differences between WAM and the associative model; the clauses are not traversed sequentially. Instead, the try-me-else instruction associatively selects the clause-label of any of the unifiable clauses non-deterministically, selects the corresponding code area using the clause-label, saves the clause-label and the indices of the data area onto the control stack, releases the flag for the corresponding clause from the clause-filter-vector, and jumps to the code area.

5.3 Handling shared-variables

In a conjunctive goal (or subgoals), the first occurrence of the variables may be bound to a vector of values. In such cases, the indices of the corresponding binding-vectors are saved in the heap with the corresponding variables. The values from the vectors are selected one at a time, and placed in a global register, and the selected binding is released from the binding-vector. This process is continued using an iterative loop until the binding-vector is null, and deep backtracking occurs. The instruction set to handle the repetitive test for backtracking is

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

The first instruction stores the <label> onto the control stack along with the other information, and goes to the next instruction to pick up a value from the binding-vector. Upon backtracking, the control comes back to the instruction “next-value”, and next iteration takes place.

5.4 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 procedures must be solved separately to derive the 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 logical vector. 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, the complement of the bit-vector corresponding to these clauses is logically ANDed to prune the processed clauses; the next clause from the other procedure is picked and processed.

6 Conclusion

In this paper, we described an associative data-parallel compilation model of logic programs capable of answering queries with unspecified relations about the given objects. The model benefits from the synergy resulting from associative search, data-parallelism during goal reduction, and the use of low level code to invoke the subgoals and savings in data-transfers due to the presence of global registers. The use of associative tables extends the power of logic programming to answer a large class of queries to derive unspecified relation-names for the given objects. In contrast to the interpretation based on pure data parallel model [2, 3], 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. Currently, the model does not incorporate user defined vectors and associations. In the future, we intend to enhance the capability of the model to manipulate user defined sets, sets of tuples, bags, and sequences in an associative data-parallel manner.

Acknowledgements

We acknowledge Madhavi Ghandikota for implementing the emulator for the model [5].

References


