ENHANCING UNIFICATION IN PROLOG THROUGH CLAUSE INDEXING

ROBERT M. COLOMB

▷ An implementation of PROLOG based on general clause-indexing methods using bit-serial content-addressable memory hardware is presented. The approach permits extremely high-performance implementations of knowledge-based applications and enables a greater reliance on unification in PROLOG programming. The indexing methods are easily attachable to other implementations of PROLOG. Three hardware-implementable indexing methods using *m*-in-*n* coding or superimposed coding are described and compared, and a bit-map representation of the set of clauses responding to a goal is described. Some benchmarks are given, and a group of useful unordered descriptor primitives described. The implications for programming style are discussed. ◁

1. INTRODUCTION

Unification is a major feature of PROLOG, distinguishing it from other programming languages. A properly designed PROLOG program will allow a query to be formulated in such a way that a response may be selected from a procedure with possibly a very large number of clauses without the programmer becoming concerned with details of how the correct clauses are identified by the system. Unfortunately, most implementations make it uneconomical to make full use of this powerful language feature. Unification is an expensive process, and if a small number of responses are selected from a large number of clauses in a procedure, the computational cost becomes prohibitive. The usual solution is to reformulate the program to reduce the size of procedures searched, gaining efficiency at the cost of increased complexity.

Address correspondence to Robert M. Colomb, CSIRO Division of Information Technology, Box 1599, Macquarie Centre, North Ryde, NSW 2113, Australia.
Received May 1988; accepted February 1989.

THE JOURNAL OF LOGIC PROGRAMMING

655 Avenue of the Americas, New York, NY 10010 0743-1066/91/$3.50

This paper reports a PROLOG-based on hardware implementation of general clause-indexing methods. The indexing techniques, based on *m*-in-*n* coding and superimposed coding, make it practical to have procedures with tens of thousands of clauses, which may even be stored on disk. An adaptation of the indexing technique permits unification based on unordered groups of descriptors, thereby increasing the practical utility of unification and providing the potential for a clear and elegant programming style.

In the following sections, the clause-indexing problem is discussed and the general techniques of *m*-in-*n* coding and superimposed coding are presented, with a sketch of the hardware implementation. Three indexing methods, all of which are implementable using the hardware, are then described and compared. Test results are presented from an experimental implementation, and its integration with UNSW PROLOG is sketched. Language extensions to effectively manipulate unordered groups of descriptors are then described. Some implications for programming style are discussed; then a summary and conclusion completes the paper.

2. THE CLAUSE-INDEXING PROBLEM AND ITS SOLUTION

2.1. Introduction

Much of the computational effort in a PROLOG program is consumed in the process of unification. In problems involving large data/knowledge bases, unification becomes a much greater proportion of the computation, as it must include the effort of retrieval of data residing on slower secondary storage. Since unification conceptually requires a linear search of the data, techniques are needed which index the set of clauses. They will either permit sections of the procedure to be bypassed or will allow a fast search which eliminates most failing clauses cheaply. It is important that no matching clause be missed by the indexing procedure.

Indexing for unification differs from conventional database indexing in that in general a PROLOG clause can have an arbitrarily structured set of arguments, and also can contain variables which match all candidates in the particular argument position.

Most PROLOG implementations follow [22, 23] by indexing on one or more of the principal functor of the clause; the number of arguments of the principal functor; whether the first term is a constant, variable, list or structure; or the first few arguments of the principal functor. These ad hoc indexing methods can be useful in problems of moderate size, but in many cases a procedure with very many clauses can be uniform with respect to the indexing method, so effectively reverting to a linear search. The most common of these methods is indexing on the first argument of the procedure using hashing methods, and will be referred to as *first-argument indexing*.

MU-PROLOG features an integrated relational database manager [14] using a hashing technique for database access [15]. Only clauses without structure or variables may be indexed. A related implementation extended to terms with variables [3].

Earlier approaches are described in [9, 18].

The MU-PROLOG hashing scheme is analysed in detail in [5], and shown to have serious problems which make it not suitable as a basis for general clause indexing. Its main problem is that unless the database has quite particular statistical properties, it will tend to return a large number of clauses as potential matches which will fail in subsequent unification. The problem seems to be fundamental to this class of methods. Furthermore, it does not lend itself well to matching variables or structured terms.

Several indexing methods have been published which are based on bit-matrix representation of the clauses in a procedure. They are *field encoding* [24], *superimposed coding* with imbedded position and variables (PROLOG-SCX) [6], and superimposed coding with external variables (NU-PROLOG) [16]. They are all based on the principle of m-in-n coding. These three methods are described in detail below, and compared.

2.2. *m-in-n Coding*

In m-in-n coding, the value of an attribute is represented in a binary word of width n by setting a fixed number m of bits to 1. This number m is called the *weight*.

The number of values representable with m-in-n coding is the number of combinations of n things taken m at a time, which is maximum when m is half n (half the bits in the code word are 1). The information (in bits) representable in a code word of width n and weight $n/2$ is

$$\log_2\binom{n}{n/2}.$$

The information representable in code words of several convenient widths is presented in Table 1.

The advantages of m-in-n coding are:

The code word is compressed, reducing the size of the tables to be searched.

The index can be searched using bit slice techniques, which reduces the number of bits to be examined in the search.

The technique is easily implemented in hardware, thereby making the searches extremely fast for tables of moderate size.

Figure 1 shows a table with a 3-in-6 encoding. A query on the table would be to identify the entry having the value *red*. Since the number of 1 bits in each of the code words is the same, it is sufficient to AND together the bits in each code word where the query code word has a 1. If the product is 1, then all of the bits must be 1 and the code word equals the query word. If the product is 0, then at least one of the bit positions in the code word had the value 0; therefore the code word is different from the query word. In this example, only half the number of bits need be searched. From Table 1, we see that a 3-in-6 encoding can represent 4.5 bits of information, so that there is a reduction in the number of bits to be searched. The reduction gets larger as the width of the code word increases.

If we wish to represent by a bit map the set of entries in the table matching the query, we can simply AND together the three indicated columns, which is an operation easily performed by bit-serial content-addressable memories.

TABLE 1. Number of Bits in an $n/2$-in-n Code Word

n	Bits
2	1.0
4	2.6
6	4.3
8	6.1
10	8.0
12	9.9
14	11.7
16	13.7
32	29.2
64	60.7
128	124.2
256	251.7

There are two basic techniques employed for encoding attributes in this scheme. If there are a small number of possible values all of which are known in advance, it is simple to generate a table of possible codes to be stored in parallel with the table of possible attribute values. Where the number of possible values is large or not known in advance, it is usual to employ a random-number generator to calculate the bit positions to set to 1. It has been found adequate to use cyclic shift-register techniques, which are easily implementable in hardware [6, 8].

Superimposed coding is a compact and efficient way of combining the codes for several attributes. Instead of each attribute having its own field, the attributes are encoded with a sparse encoding (e.g. 5-in-64), and the encodings for the several attributes in the same clause head are ORed together (hence superimposed). The coding parameters are chosen so that the average encoding of all attributes in a clause head will have half the bits set in its code word.

The cost involved in superimposed coding is the possibility of *false drops*, where the index search indicates a matching clause but the clause does not, in fact, match. All hashing-based indexing methods have this problem due to aliasing, but the problem is compounded in superimposed coding due to interference between the encodings of different attributes. In principle, the problem can be minimized by choice of coding parameters based on the number of clauses to be indexed, the number of attributes to be indexed per clause, and the number of constants

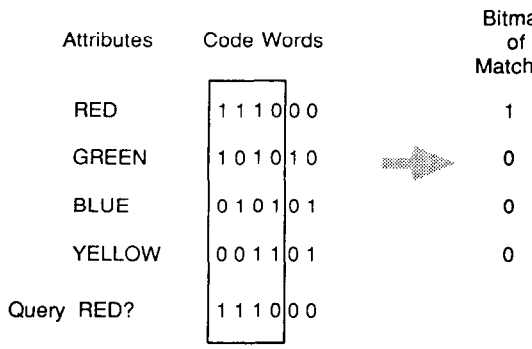


FIGURE 1. Example of m-in-n coding.

present in a goal. The problem is discussed in detail elsewhere [6, 17, 20, 21], and some choices are given in the benchmarks below.

2.3. *Hardware Assistance for Indexing*

In the previous section, it was noted that the elementary operations needed for query processing in an *m*-in-*n* coded index can be performed by a bit-serial content-addressable memory. One such device, the *relational algebra accelerator* (RAA) [10, 11], can perform a logical operation on two 4096-bit columns in 40 microseconds, giving an average processing rate of 100 million bit operations per second. For comparison, a 1-Mips 32-bit machine in a six-instruction loop can process 5 million bits per second. A second version (RAA-2) will be completed shortly which will be able to process two 16-Kbit columns in 10 microseconds, for a processing rate of 10^9 bits per second. A third version (RAA-3) has been designed and prototyped in VLSI. It has a different architecture, and is expected to have a cycle time of less than 100 nanoseconds. An RAA-3 with a column size of 64 Kbits is quite feasible, and would process at a rate of about 10^{12} bits per second. Since these devices are inexpensive (an RAA with two banks of 256 4-Kbit columns retails at less than $US2000, while an RAA-3 with 256 65-Kbit columns would cost about $25,000), these bit-matrix indexing schemes become very attractive.

A content-addressable memory gets its performance from high-speed access to data contained in it. If it is necessary to transfer data from some other storage device, such as conventional RAM or disk, the transfer time is so high that a very large number of searches must be made on the data before the transfer time ceases to dominate the total computation. A content-addressable memory is therefore advantageous mainly in problems where the indexes can reside semipermanently in its memory. The RAA has two banks of 256 4096-bit columns, so is capable of storing the indexes to about 8000 clauses. The other versions mentioned are larger, but it would appear to be uneconomic to store more than perhaps 100,000 clauses. We therefore see the techniques being used to index knowledge bases rather than databases.

2.4. *Basis for Comparison*

The three methods will be compared on a number of dimensions:

Coding efficiency: Given a fixed index word width, the percentage of the information carrying capacity of that word potentially used by the indexing method.

Flexibility: How rigidly it is necessary to specify the structure of a procedure in order to make use of the technique.

A table to be indexed can be resolved into equivalence classes within which the entries are not distinguished by the indexing method. Discriminating power comes partly from the information content of particular attributes and partly from the number of attributes combined in a query. If the *strength* of an attribute is measured by the extent to which it alone resolves the table, a *strong attribute* is one which will on the average identify a single entry. A single entry may equally

well be identified by a combination of *weak* attributes. It sometimes happens that a table is *overdetermined* in that there are several strong attributes. The *discriminating power per attribute* is the strength of an attribute that can be represented in the index. If the discriminating power per attribute is weak, a strong attribute will have many values represented by the same value in the index (aliasing), and so will be weakened.

Cost of representing variables (space): Additional bits in the code word needed to represent variables.

Cost of representing variables (time): Additional processing time needed to handle the possibility of variables in clause heads.

Encode position of constants: Whether the method discriminates between the same constant in different positions in the clause head.

The methods are evaluated in the following, and the evaluations are summarized below in Table 2 (Section 2.8).

2.5. Field Encoding

Field encoding solves the problem of structure by requiring the programmer to specify a template for the clauses in an indexed procedure. For example, all the clauses might be of the form

$$f(A, G(B, C), H(D)).$$

A total width for an index word is selected, and it is divided into fields so that each possible constant in the clause head is assigned a separate m-in-n encoded field. The width of each field is up to the programmer. An easy way to allocate fields is to count the possible constants and divide the index word width by that number. The above example has six positions for constants. If the index word were allocated 36 bits, each field would have a width of 6 and weight of 3, as shown in Figure 2.

Variables in a clause head are indexed by simply setting all the bits in the corresponding field to 1. The resulting field will match any pattern of bits, and so

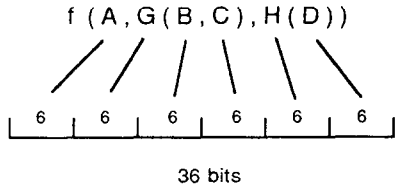
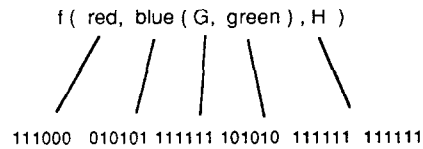


FIGURE 2. Example of field encoding.



will always respond, as indicated in Figure 2. The encodings are shown from the example in Figure 1. Where a variable appears in a position occupied by a structure, all bits allocated to the structure are set to 1. The encoding of the variable H in the figure is an example of a variable in a position occupied by a structure in the template.

The coding efficiency of field encoding is reduced by the use of separate fields to represent structure. In the example above, each field can contain 4.3 bits of information (Table 1). The six fields together can contain $6 \times 4.3 = 27$ bits, or 72% of the 36 bits allocated to the index word. An extreme example is a procedure with eight arguments, each of which can be a function with seven arguments. There are 64 possible constants. If the index-word width is 256 bits, there will be $256/64 = 4$ bits per constant. From Table 1, a four-bit-wide field can contain 2.6 bits of information. The indexing method allows the 256-bit field to carry $2.6 \times 64 = 166$ bits of information, for 65% efficiency. (Note that the inefficiency comes entirely from the inefficiency of m-in-n coding for smaller fields.)

The flexibility of the method is low, since if the pattern of arguments deviates from that anticipated by the programmer, the bits devoted to representing the missing structures are lost.

The discriminating power per attribute tends also to be low. In the examples given above, the individual fields can take only 20 (3-in-6 encoding) or 6 (2-in-4 encoding) different values. In order for the method to cope with an overdetermined table, the index word must be very much wider.

The space and time costs of representing variables is nil, since the coding scheme ensures that any constant will match a clause head containing a variable in that position or any higher position.

Finally, the technique allows the position of constants to be represented.

2.6. Superimposed Coding; External Representation of Variables

The superimposed-code clause indexing system in NU-PROLOG is implemented as a disk file access technique, but the underlying representation technique is adaptable to a medium-size problem which can be implemented in hardware. Like field encoding, this scheme uses a template for the clause heads in the procedure to be indexed, but instead of allocating discrete fields to each attribute, the attribute code words generated by the first level of structure are superimposed using a fixed weight. If the template includes substructure, as illustrated in Figure 3, the attributes at a lower level are encoded with a weight which is a proportionate share of the weight of the code word at the parent level. (In the figure, the positions B and C are encoded with three bits, while the others are encoded with six.) Only constants are encoded. Any terms outside the template do not participate in the index. In indexing

$$f(a(x), g(b, C), h(d), y).$$

using the template in the figure, only the constants a, g, b, h, and d would be encoded. The constants x and y would be omitted because they are outside the template, while C would not be encoded because it is a variable.

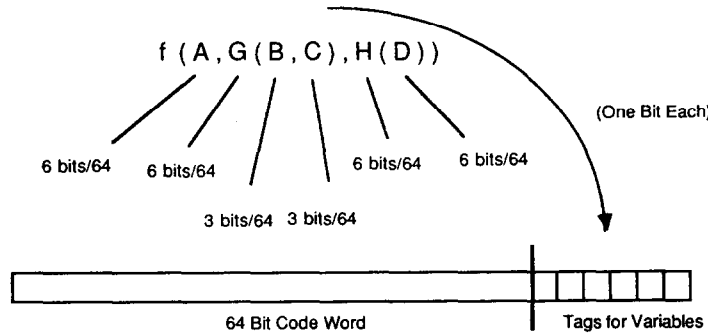


FIGURE 3. The NU-PROLOG coding scheme.

Variables are represented in a separate tag field very like the field-encoded index template with a field width of one bit. The variable C in the previous example would cause the sixth bit in the variable tag field to be set.

A query against a procedure guaranteed to contain no variables is processed very simply. The query is encoded in the same way as the clause heads; then the bit map of responding clauses is calculated using the normal methods of superimposed coding.

When the procedure may contain variables, the query processing is more complex. The goal term must be processed one term at a time. A particular clause in the procedure matches the goal if the constant matches, or if a variable in the position of the constant matches, or if a variable in any position higher in the structure matches.

Consider the goal structure $a(a_1, \ldots, a_n)$. If, for a particular clause, we represent by A the proposition

"The functor a matches a functor in the clause head",

by A_i the proposition

"The constant a_i matches a constant in the ith argument position in the clause head",

by V the proposition

"The functor a matches a variable in the clause head",

and by V_i the proposition

"There is a variable in the ith argument position of the clause head",

then the proposition "$a(a_1, \ldots, a_n)$ matches the clause head" is calculated as the boolean-algebra expression

$$V + A \,\&\, (V_1 + A_1) \,\&\, \cdots \,\&\, (V_n + A_n), \tag{1}$$

where + denotes inclusive OR and & denotes AND. The formula is recursive, with A_i replaced by a similar formula if the term a_i is structured. This formula involves each symbol only once for an expression of any depth.

The A predicates are evaluated using the superimposed code word, while the V predicates are evaluated using the variable tag field.

Superimposed coding is efficient: a 128-bit field can be seen from Table 1 above to be up to 97% efficient.

The NU-PROLOG scheme is more flexible than field encoding. Although a procedure template is used in the same way, the more efficient superimposed-code indexing imposes less of a penalty when clauses with less structure than the template are indexed.

The discriminating power per attribute is very high at the first level of argument, and still good at the second (a 2-in-128 code has 8128 possible values, for example).

Variables are represented with a space cost of one bit per attribute position in the template, which is moderate, while the time cost is also moderate, involving searching one bit position per constant in the query.

The technique does not encode the position of constants in a structure.

2.7. Superimposed Coding with Position and Internal Variables

Another version of superimposed coding, called PROLOG-SCX, is built on an indexing scheme first reported in [8]. It represents variables and also structure.

The key difference from the NU-PROLOG scheme is in the representation of structure. It is done essentially by *naming* the position in a structure occupied by each term, then *encoding the name*. In this way, structural position is expressed similarly to constant values. A variable in a particular position is represented simply by encoding the position. (This is analogous to the marking of a position in the other schemes.) A constant in a particular position has two identifiers, its value and its position. The two identifiers are combined into one before encoding (analogous to the location of the encoded constant in a particular part of the field encoded word, described above). Because position is represented at the level of names, the encoding process is similar for all terms.

A second key element in PROLOG-SCX is the representation by a bit map of the set of clauses responding to a goal. The outcome of a query on a bit-slice-organized superimposed code index is naturally a bit map containing 1 in each position corresponding to a matching clause. Complex queries may be reduced to logical combinations of simple queries, and the logical combinations are easily calculated as binary logical operations on the bit maps resulting from the simple queries. This bit-map representation arose from a focus on hardware implementation of the indexing scheme, and is applicable to the other methods.

A position is named by the tree address of the item in the clause head. For example, the variable X in the term $f(a, b(c, d(X, Y)))$ has position $(2, 2, 1)$, since it is the first argument of $d(X, Y)$, which is the second argument of $b(c, d(X, Y))$, which is the second argument of the term. This name is represented by a sequence of bytes. A 32-bit word can hold four bytes of tree address.

The index is controlled by a weak form of template. It is necessary to specify the maximum depth of structure to index. It is implicitly necessary to estimate the number of attributes to index, as well, to calculate the index parameters, but the attributes in a given clause head can be distributed anywhere within the nominated depth of structure.

A simple example may help to clarify the method. We index the term $f(a, X)$. Let $hash(x)$ be a hashing function mapping a text string into a 32-bit word. The

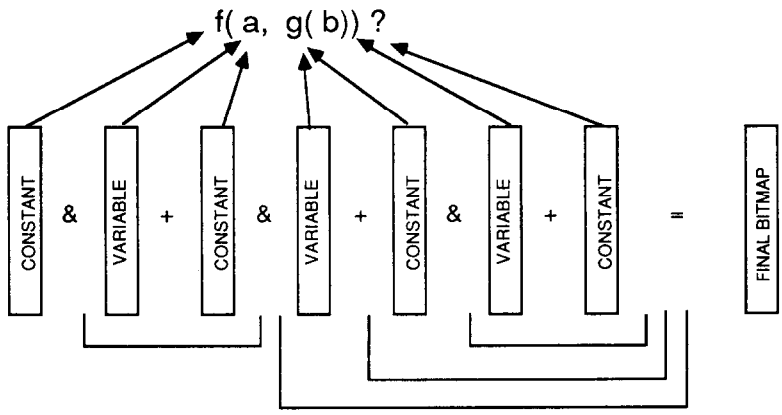


FIGURE 4. Bit-map operations for calculation of bit map for clauses responding to query. Variables allowed in procedure.

function used in the experimental implementation is a simple exclusive OR. Let *encode*(*y*) be the result of performing the superimposed code encoding of the 32-bit word *y*.

The superimposed-code representation of $f(a, X)$ is

$encode(hash(f)) +$	{*principal functor*}
$encode(hash(a) \wedge (1,0,0,0)) +$	{*constant*}
$encode((2,0,0,0))$	{*variable*}

where (i, j, k, l) is the 32-bit word whose successive bytes are i, j, k, and l. The operator $+$ is bitwise inclusive OR, while $\wedge$ is bitwise exclusive OR.

Evaluation of one of the predicates in the underlying indexing scheme is based on bitwise AND operations and produces a bit map of clauses matching that term. Evaluation of a query against a procedure containing variables mirrors exactly the boolean expression (1), using bitwise AND and OR operations on the bit maps, as illustrated in Figure 4. Although somewhat more complex than the nonvariable case, it requires examining only twice as many bit positions.

A total of seven bit maps must be calculated in the example of Figure 4, one for each of the four constants in the query, and one for the possibility of a variable in each of the three argument positions. Calculation of each bit map requires searching a small portion of the index. For example, if a 5-in-64 encoding is used, each bit map will require processing 5/64 of the index, or 7.8%. Calculation of all seven bit maps will therefore require processing 55% of the index. (A query on a procedure in which variables are not permitted in clause heads would need four bit maps, so it would require processing 31% of the index.) The bit maps must then be combined using AND and OR operations as parenthesized in the figure.

The efficiency of the technique is high, since it is based on superimposed coding.

It is also very flexible. The depth of structure to index and the average number of attributes per clause head are the only restrictions.

TABLE 2. Comparison of Indexing Methods

Characteristic	Field	NU	SCX
Coding efficiency	Medium	High	High
Flexibility	Low	Medium	High
Disc. power/attribute	Low	High/med.	High
Cost of var. (space)	Nil	Medium	Nil
Cost of var. (time)	Nil	Medium	High
Position of constants?	Yes	No	Yes

The discriminating power per attribute is also high. At the top level it is the same as the NU-PROLOG method, while below the top level it is very much higher, since each attribute can be indexed with the same number of bits.

The cost of variables in space is nil, since variables are indexed in the same space as constants; but the cost of variables in time is high, since they are represented with several bits.

The technique encodes the position of constants within structures.

2.8. Comparisons

Table 2 gives a comparison among the three methods on six characteristics: coding efficiency, flexibility, discriminating power per attribute, space and time cost of variables, and whether it can distinguish constants in different positions. The methods are called Field (field encoding), NU (NU-PROLOG scheme), and SCX (PROLOG-SCX scheme).

Each of the methods has advantages in certain applications. Field encoding is suitable for a procedure with a simple regular structure and not too many attributes, especially if it is not overdetermined or if there are a large number of variables (a decision table, for example). The NU-PROLOG scheme is superior for applications where the order of attributes is not important or different attributes are drawn from different populations of constants, or where the structures are moderately simple and there are variables. The PROLOG-SCX scheme is superior where structures are complex (e.g. trees) or irregular, or where many attributes are drawn from the same population of constants and field encoding is not suitable.

3. TEST RESULTS

The superimposed-code clause indexing scheme is intended to be used for those procedures in a PROLOG program which are datalike. That is, either the procedure has a large number of clauses, functioning as a knowledge-base analog to a file, or the procedure is one of a (possibly large) group of (possibly small) procedures which are volatile: subject to frequent *assert*s and *retract*s. These datalike procedures either have a permanent existence or are used as working memory in extensive computations. The datalike procedures in a PROLOG program are syntactically similar to the programlike procedures, which are small and fixed. They are distinguished only by their size or patterns of use.

Because datalike procedures are similar to programlike procedures, the implementation of the indexing scheme is nearly completely transparent to the programmer. This makes it easy for a program to be developed with a few clauses in the datalike procedures, without indexing, then converted to a production version with indexed procedures when convenient.

A procedure can either be indexed separately (if it is a large filelike procedure), or together with other procedures. Associated with each index is a control block giving indexing parameters. Procedures belonging to the same index perforce have the same indexing parameters. If a procedure is indexed separately, the index control block is identified by its principal functor. A separately identified index control block is specified for all other procedures together.

One of the key properties of superimposed code indexing is that it is highly compressed. There is therefore a range of procedure sizes where the procedure must reside on secondary storage but the index can reside in memory. This means that both the clauses and the index may reside either in memory or on disk, in three combinations:

(1) both index and clauses in memory;
(2) index in memory, clauses on disk;
(3) both index and clauses on disk.

Boundaries between the cases are arbitrary and depend on the amount of content-addressable memory available for the index and the amount of real memory available for the clauses.

PROLOG-SCX presently implements the first two cases, where the index resides in memory, thereby exploiting the real strength of the hardware implementation. The UNSW PROLOG has the index in content-addressable memory, and clauses in (virtual) memory.

In the third case, where the index must reside on disk, superimposed coding is shown in [5] to be not so generally superior a method, although, given an ideal hardware and operating-system substrate, superimposed coding can be effective for up to about 100,000 clauses. The two-level indexing scheme used in NU-PROLOG is better adapted to this situation, which is typically a databaselike environment.

PROLOG-SCX is therefore oriented towards problems where the number of clauses is up to the tens of thousands, but not to a full commercial database, which may have millions of records. Knowledge bases are typically in the range of sizes handled by PROLOG-SCX. For example, the expert system R1, a particularly large one, has fewer than 4000 rules [13].

When an indexed procedure is encountered as a goal, index searching is performed as described above, and the resulting bit map of potential matches is stored in the *control stack*, along with a count of the number of potential matches not yet used and a pointer to the last match used. On backtracking, the bit map is traversed without reference to the index for the procedure. This implies that a query to an indexed procedure will return all and only those solutions which were valid at the time of the initial goal invocation, irrespective of subsequent *assert*s and *retract*s.

In this respect the approach taken is similar to that taken by Chakravarthy et al. [2] and differs from that taken in MU-PROLOG by Naish and Thom [14]. In the latter, a goal invocation sets up a process which traverses the database, one step

each time the goal is backtracked to. No goals may be active in a procedure when a change is made via *assert* or *retract*.

Results are presented from an implementation of the indexing schemes loosely imbedded in UNSW PROLOG [4] running on a Sun 3/160 with an attached RAA, under the UNIX operating system. The indexing primitives are implemented in C and accessed by PROLOG through a variation on the *clause* builtin. Five test cases are presented:

Wise/Powers-4K. Relation with 4096 clauses of the form

stock(*base*(*brown*,*trim*(*purple*,*nightshirt*)),

base(*velvet*,*trim*(*cotton*,*nightshirt*))).

using the query

stock(*X*,*base*(*leather*,*T*))?

which had exactly one responding clause. (Adapted from [24].) Indexed by a 4-in-64 encoding of the first two levels of argument, 10 items per clause, using the SCX method of indexing with position included, but no variables allowed in the clause heads.

Decision Table-4K-4. Simulated decision table with 4096 clauses of the form

dectab(*alpha*,*omega*, _,*alpha*,*alpha*,*omega*, _, _,*omega*,*omega*).

Generated by taking every 15th occurrence of cycling through 10 tables of (*alpha*,*omega*, _) with some adjustment. The query used was

dectab(*alpha*,*omega*,*alpha*,*omega*,*alpha*,

omega,*alpha*,*omega*,*alpha*,*omega*)?

which had one clause responding. Note that this procedure has variables. It was indexed under the SCX method by a 4-in-64 encoding of all 10 arguments. The query produced 255 false drops.

Decision Table-4K-6. Similar to Decision Table-4K-4, except indexed by a 6-in-128 encoding. Only one clause responded.

Field Encoding-4K. The decision table above, indexed by a field encoding scheme with ten two-bit-wide fields. Only one clause responded.

Descriptors-4K. The procedure contains 4096 clauses, each of which contains a list of descriptors (words drawn from a vocabulary of 4742 words, derived from a mailing-list application). The lists average 14 descriptors each. The coding scheme derived from the NU-PROLOG method, in that the entire list is encoded using a 6-in-128 code without regard for position. No variables are allowed. The query is of the form

find([*mr*, *aitkenson*, *wa*, *industry*], *X*)?

where the responding clause contains the query's list of descriptors as a subset of its list of descriptors (without regard for order). Only one clause responded.

TABLE 3. Results of Test Cases

Test	Unif. attempts (10^3/sec)	Time/success (msec)	False drops (%)	Bits searched	Time searching (msec)
Wise/Powers-4K	146	28	0.0	8	0.32
Decision Table-4K-4	11	372	6.2	40	1.60
Decision Table-4K-6	68	60	0.0	60	2.40
Field Encoding-4K	114	36	0.0	10	0.40
Descriptors-4K	91	45	0.0	24	0.96

Table 3 shows results of experiments on the test cases described above. "Unif. attempts" is the number of unifications attempted per second (in thousands), obtained by dividing the number of clauses in the main procedure by the time taken to find a matching clause at the end of the procedure. In these benchmarks, this measure is roughly comparable to the number of logical inferences per second (KLIPS) in benchmarks such as Naive Reverse. "Time/success" is the number of milliseconds per successful unification, computed by dividing the total execution time by the number of successful indexed procedure unifications. "False drops" is the percentage of the total number of clauses in the indexed procedure which responded to the indexing search but which later failed unification. "Bits searched" is the number of bit positions searched on the RAA. "Time searching" is the number of milliseconds spent on the RAA performing the index search.

The table shows that hardware-supported clause indexing is effective, with an average searching power equivalent to about 100 KLIPS. This is true with and without taking into account the position of terms and with and without allowing variables in the clause heads. It also shows that all of the indexing schemes are effective in appropriate cases. The one poor performance, that of Decision Table-4K-4, is poor because of the number of false drops (255) caused by allowing insufficient weight in the superimposed code index, a defect corrected in the following Decision Table-4K-6. Even so, its performance of 11 KLIPS is not disastrous, considering that UNSW PROLOG is rated at about 2 KLIPS on the Sun 3/160.

Note that in all cases the index is held in memory. For the largest case, Descriptors-4K, the superimposed code index took up 64 Kbytes. The next largest case was Wise/Powers-4K, for which the superimposed code index took up 32 Kbytes. The bit map of responding clauses was 512 bytes in all cases.

The "Time searching" column in Table 3 shows that there is considerable room for optimization in the integration of the RAA with PROLOG. Benchmarks running on the same equipment not imbedded in PROLOG give a performance of 40 microseconds per bit position searched (from which the time-searching value is derived). The remainder of the time is spent in setting up data structures, etc. (which could be much reduced if the indexing method were well imbedded in a good compiled PROLOG) and in aspects of the indexing other than searching the bit matrix. These aspects include calculation of the superimposed code word for the query (except in Field Encoding-4K), and in searching the resulting bit map for 1 bits.

TABLE 4. Results of Test Cases with Estimated Effect of First-Argument Indexing

Test	Unif. attempts (10^3/sec)	Time/success (msec)
Wise/Powers-4K	341	12
Decision Table-4K-6	95	43
Field Encoding-4K	228	18
Descriptors-4K	146	28

In particular, about 20 milliseconds in each case is spent in finding the responding clause at the end of the 4096-clause procedure. Somewhat less than 3 milliseconds is spent in searching the bit map, and the remaining 17 milliseconds is spent in chaining through the clause table, since UNSW PROLOG does not have first argument indexing. To show that first argument indexing is complementary with the kinds of indexing reported in this paper, the key columns of Table 3 are repeated in Table 4, showing an estimate of the times taken for the four cases considered if first-argument indexing were available. The figure for time/success is that from Table 3 reduced by 17. This change increases the average performance to about 200 KLIPS.

Several other optimizations are not difficult. Calculation of the query superimposed code word by the method used takes about five times as long as performance of the index search. The random numbers needed are generated by an algorithm involving cyclic shift registers, and so may be easily computed in hardware. Alternatively, in Decision Table-4K-6 and Descriptors-4K the superimposed code words can be computed at compile time. Further, as mentioned above, searching for 1 bits takes up to 3 milliseconds if the bit found is at the end of the 4096-bit table. This function is also easy to implement in hardware. In fact, it is intended that the RAA-2 have both these features.

In short, a combination of software and hardware optimization should be able to increase performance by a factor of five or more. In addition, it is possible to share the overhead by holding more than one table in the RAA (it has the capacity to hold four 4096×127 bit tables). A 16-K clause procedure could be processed in considerably less than four times the time required for one. Further, several RAA boards can be placed on the same system, permitting a larger column size with the same processing time.

The PROLOG-SCX system has, as well as the indexing scheme, an implementation of a persistent PROLOG environment held on disk [6, 7]. It is clear from Table 3 that the hardware-assisted indexing scheme is fast enough so that if the indexed procedure were held on disk rather than in memory, performance would be dominated by the time needed to retrieve the selected clauses from disk. It would be especially important in this case to select the coding parameters so as to minimize the number of false drops.

Performance would be affected by holding the clauses on disk. In the worst case, an additional penalty of say 20 msec would be paid per clause responding to the goal, plus the same for each false drop. In practice, most operating systems read a large block into a cache buffer. Assuming the clauses in a procedure are stored together, the time per access is therefore greatly reduced, and is comparable to

page faulting in a virtual-memory system. In addition, if some information about access patterns is known, it is possible to cluster the clauses on disk to increase the density of responding clauses per disk block, by methods such as used in NU-PROLOG [16]. If these methods are appropriate, performance superior to virtual memory can be expected.

The indexing schemes described above make programming with large procedures practical, especially when the access time per clause is slow due to the clauses being held in secondary store, whether explicitly or using virtual memory. Clearly, under these conditions it is essential to employ a coding scheme which minimizes false drops.

4. IMPLICATIONS FOR PROGRAMMING STYLE

In order to use the content-addressable memory effectively, it is important that the indexed procedures contain as much pattern description as possible in the clause heads. Assume a problem containing a procedure with a large number of clauses which is central to the computation to be performed. The processing cycle is to get some parameters from the input, then select one of the clauses in the large procedure to do the processing. A suitable programming style will produce the following:

get_parameters(*Identifying_parameters*, *Input_parameters*),

find_clause(*Identifying_parameters*, *Responding_clause_id*),

compute(*Responding_clause_id*, *Input_parameters*, *Output*)?

The procedure *get_parameters* obtains two sets of parameters: *Identifying_parameters*, which are used to determine which clause is to do the processing, and *Input_parameters*, which provide further input to the processing. The procedure *compute* takes the input parameters and produces the output values in the set of variables designated *Output*. The intermediate procedure *find_clause* is indexed and does a partial match search using *Identifying_parameters* to identify the proper processing clause, binding its identifier to the variable *Responding_clause_id*. The procedure *compute* is also indexed, but only on the single argument *Responding_clause_id*. The procedure *find_clause* has only the parameters necessary to identify the action clause, thereby making efficient use of the indexing. This approach is especially attractive in a PROLOG with first-argument indexing.

A program like the above will be executed much more quickly than a more conventional program like

get_input(*Input_parameters*), *do_p*(*Input_parameters*, *Output*)?
do_p(*Input_parameters*, *Output*):-
  validate_input(*Input_parameters*),
  calculate_output(*Input_parameters*, *Output*).

where the goal *validate_input* acts as a guard to the computing procedure *calculate_output*, but the unification with the clause heads of *do_p* is carried out repeatedly until the guard goal succeeds.

In most PROLOGs, there is little computational time difference between the two programming styles. The indexing system described so far makes the former much faster than the latter for a large class of problems. However, in many cases the *validate_input* procedure would do something like check that a constant supplied as an input parameter is an element in a list of descriptors held in the head of the clause being considered. Since the *member* test operates without regard for the order in which the elements are found in the list of descriptors, it is not possible to perform this test as part of unification. Further, the *member* test is expensive, as several unifications may be needed to evaluate the predicate.

5. UNORDERED DESCRIPTORS

5.1. Language Extensions

As shown by the example Descriptors-4K in Table 3 above, the indexing scheme used in NU-PROLOG can be easily adapted to perform an index search for predicates like *member*. To mobilize this resource, a group of *set* builtins has been added to PROLOG-SCX, and a shorthand syntax has been added to permit unordered descriptors to be combined with the structured terms in a way which makes no change in the logical behavior of the PROLOG program.

An unordered group of descriptors can be listed in a builtin predicate *set_of*, which has the semantics of an elementary set. A set is defined as

⟨*set*⟩ ::= { } | {⟨*elements*⟩}

⟨*elements*⟩ ::= ⟨*ground_term*⟩ | ⟨*ground_term*⟩, ⟨*elements*⟩

where ⟨*ground_term*⟩ is a normal PROLOG ground term. (Since a set has by definition unique elements, it is hard to see a semantics for allowing terms with variables.) A set is therefore a subset of the Herbrand universe.

Evaluable predicates are provided to support the normal set operations, using the assignment operator *is*:

*S*1 *union* *S*2

*S*1 *intersect* *S*2

*S*1 *difference* *S*2

G *member_of* *S*

*S*1 *contained_in* *S*2

*S*1 *set_equals* *S*2

card(*S*)

plus one additional to support signature-analysis-type applications:

minimum *N* *of* *S*1 *contained_in* *S*2

which is true if *N* is an integer, *card*(*S*1 *intersect* *S*2) >= *N*.

Space does not permit a complete description of the implementation, which may be found in [6]. The key point is that, for the convenience of the programmer and, more important, for ease of implementation of the superimposed-code index

searching, the following may appear as shorthand in goals:

contained_in

member_of

minimum N of ... contained_in

in the form

p(*X contained_in S*)?

Note that *member_of* may appear in two ways: either the element or the set can be ground.

In all the above cases where the shorthand appears in a goal in an indexed procedure, the goal will be first subject to index searching via the method of superimposed codes.

In the above shorthand the predicate is applied during unification. This is logically equivalent to inserting the set predicate as a goal immediately after the responding clause head.

The *set of* construct $\{X_1, X_2, \ldots, X_k\}$ is implemented as a k-ary function

set of$(X_1, X_2, \ldots, X_k)$.

The functor *setof* is an evaluable predicate.

When *setof* appears in unification, the order of elements is ignored. Two sets will unify if they are equal in the set-theoretic sense.

Sets are assumed to have unique elements. The explicit constructor predicate *union* will not add any duplicate elements. The present implementation, however, does not check for duplicate elements in sets described by example.

Since the elements of a set are unordered, the system of encoding terms described earlier for PROLOG-SCX is not directly applicable. The set itself has a position in its predicate. Each element term is encoded by exclusive-ORing together all its hashed constants without regard for their structural relationships, then combining with the position of the set. A superimposed code word is then calculated for each term.

This method of encoding assumes that there is not a great deal of structure in the terms appearing as elements in sets. It does not distinguish $f(a, b)$ from $f(b, a)$, nor $a(b)$ from $b(a)$, for example. It does, however, distinguish between $f(a)$ and $f(b)$. The particular method of representation is not essential. Other possibilities exist which would be better for particular applications.

The superimposed-code parameters should be set to encode successfully the number of elements expected in a set. If the code word becomes saturated, the clause containing a large set will respond successfully to index searching, but will also respond to most goals, and so contribute to false drops.

When a set appears in a goal by one of the shorthands described previously, the element terms are encoded as above and a superimposed code word calculated for the known elements of the set. This code word then participates in the query as would the code word produced by any constant.

One exception is the *minimum N of ... contained_in ...* construct. In this case, a code word and a bit map of clause heads responding are calculated for each known element. An output bit map is created with a 1 in each position in which at

least N of the individual element bit maps have a 1. This output bit map functions in further calculation in the same way as the bit map of clauses responding to an ordinary constant.

The other exception is the $p(G\ member\_of\ S)$ construct used in a goal where G is free and S is ground. This goal is equivalent to

$$p(g_1) \mid p(g_2) \mid \cdots \mid p(g_n)$$

where S is $\{g_1, g_2, \ldots, g_n\}$. The superimposed-code index search produces a bit map of responding clauses which is the inclusive OR of the bit maps for each of the goals $p(g_i)$.

Since the indexing scheme for unordered groups of descriptors is very similar to that presented in the first section of this report, the time performance in Table 3 above applies to the present scheme as well.

5.2. *Examples of Unordered Groups of Descriptors*

The set data structure described above allows (possibly more than one) unordered group of descriptors to be arguments of a clause head. As the sets are indexed and the set primitives available in goals can make use of this indexing, applications based on information retrieval (e.g. [19]) may be conveniently programmed in PROLOG.

One example of this kind of application is the retrieval of documents, where the descriptors are the major words contained in the document. Another is retrieval of titles of journal articles based on key-word descriptors supplied by an editor.

Retrieval in a descriptor-based system is based on boolean operations AND, OR, and NOT. The NOT operator is not implemented in the superimposed code index, since the coding scheme may produce a superset of the clauses responding to a goal. Use of NOT would therefore produce a subset of responding clauses.

The AND operator can be implemented using the *contained_in* or *some_of* builtins. If all objects with descriptors d_1 AND d_2 are desired, the goal

$$object(\{d_1, d_2\}\ contained\_in\ Descriptors)$$

or

$$object(some\_of(\{d_1, d_2, \ldots, Rest\}, Descriptors))$$

will perform the indexing. The construct with *some_of* will also bind the subset of descriptors other than $\{d_1, d_2\}$ to *Rest*.

An OR operation can be performed using the superimposed code indexing using the *minimum ... of* builtin. If all objects with descriptor d_1 OR d_2 are desired, the goal

$$objct(minimum\ 1\ of\ \{d_1, d_2\}\ contained\_in\ Descriptors)$$

will perform the desired indexing.

Another interesting application of descriptors arises in natural-language parsing [12]. The grammar is represented as a transition network. The object of processing is to identify a path through the network which could produce the text encountered.

Words are processed one by one. The first operation is lexical analysis. This identifies the set of possible lexical categories (noun, verb, adjective, etc.) to which the word could belong. The next stage of processing is to add to a transition network under construction. There can be several partial paths, each one to be extended if possible. If it is not possible to extend a path with the word under consideration, that path is eliminated from the set of active partial paths.

For a partial path being extended, the *present node* is the end node of the path found so far, and is the node associated with the last word analysed. The *next node* is obtained from the present node and a lexical category. In PROLOG

```
{goal}
lex_analyse(Word, Set_of_categories),
    get_node(Present_node, Set_of_categories, Next_node),
    process_edge([Present_node, Next_node, Other_parameters])?
{procedure}
get_node(Set_of_pred_nodes, Lexical_category, Next_node):-
    ⋮
```

In the goal, the possible lexical categories of the word are held in the set *Set_of_categories*. In the procedure, *Set_of_pred_nodes* is the set of predecessor nodes from which the lexical category *Lexical_category* leads to the node *Next_node*. In the form stated above, the *get_node* goal is in a different form from the *get_node* procedure. Using the set builtin *member_of*, the goal can be stated

```
get_node(Present_node member_of Set_of_Predecessor_nodes,
    Lexical_category member_of Set_of_categories,
    Next_node)
```

The first use of *member_of* has the element instantiated, so the goal is less general than the procedure. In the second use of *member_of*, the set is instantiated, so the goal is more general than the procedure. In both uses of sets of descriptors, execution is speeded by use of the indexing scheme.

6. SUMMARY AND CONCLUSIONS

This paper has presented an implementation of PROLOG based on bit-matrix solutions to the clause-indexing problem. Representation by a bit map of the set of clauses responding to a goal is the basis of the integration of the indexing scheme with the interpreter. The schemes can be fairly easily added to most PROLOGs, by adding builtins similar to *clause*.

The indexing schemes have been implemented in hardware, so they are extremely fast. Since the indexes are compressed, they can be used as well to implement persistent procedures, where clauses may reside on secondary storage. The method has a limitation that the procedures indexed may have up to a few

tens of thousands of clauses, so that it is suitable for knowledge indexing and small databases rather than large databases.

If the indexing method is fully integrated into the interpreter, it becomes possible to implement unordered groups of descriptors using *set* predicates as builtins. Goals using these predicates can be indexed using superimposed coding, permitting a great extension in the practical knowledge-representation power of the language. In addition, a full integration would further improve performance.

Further development work must be done to closely integrate the hardware accelerator with a high-performance PROLOG, and to implement additional critical functions in hardware.

The timing benchmarks were programmed by Charles Chung.

REFERENCES

1. Allen, M. W., Jayasooriah, and Colomb, R. M., AIM: A New Form of Processor and Its Application to Partial Match Data Retrieval, in: *Ninth Australian Computer Science Conference*, Canberra, 1986, pp. 347–355.
2. Chakravarthy, U. S., Minker, J., and Tran, D., Interfacing Predicate Logic Languages and Relational Databases, in: *Proceedings of the First International Logic Programming Conference*, Faculté des Sciences de Luminy, Marseille, France, 1982, pp. 91–98.
3. Chomicki, J. and Grudzinski, W., A Database Support System for Prolog, in: *Proceedings Logic Programming Workshop '83*, Algrave, Portugal, Univ. Nova de Lisboa, 1983, pp. 290–303.
4. Chung, C. Y. C., Integration of Relational Algebra Accelerator (RAA) with UNSW Prolog, Technical Report TR-FB-88-04, CSIRO Division of Information Technology, Sydney, Australia, 1988.
5. Colomb, R. M., Use of Superimposed Code Words for Partial Match Data Retrieval, *Australian Comput. J.* 17(4):181–188 (1985).
6. Colomb, R. M., A Hardware-Intended Implementation of Prolog Featuring a General Solution to the Clause Indexing Problem, Ph.D. Thesis, Dept. of Computer Science, Univ. of New South Wales, Sydney, Australia, 1986.
7. Colomb, R. M., Assert, Retract and External Processes in Prolog, *Software—Practice and Experience* 18(3):205–220 (1988).
8. Colomb, R. M. and Jayasooriah, A Clause Indexing System for Prolog Based on Superimposed Coding, *Austral. Comput. J.* 18(1):18–25 (1986).
9. Futo, I., Darvas, F., and Szeredi, P., The Application of Prolog to the Development of QA and DBM Systems, in: H. Galliare and J. Minker (eds.), *Logic and Data Bases*, Plenum, 1978, pp. 347–376.
10. Jayasooriah, An Attached Index Machine, Ph.D. Thesis, Dept of Computer Science, Univ. of New South Wales, Sydney, Australia, 1986.
11. Jayasooriah, Allen, M. W., and Colomb, R. M., Attached Index Machine: A New Form of Coprocessor for Manipulating Bit Encoded Index Structures, in: *Proceedings 2nd Australian Computer Engineering Conference*, Sydney, 1986.
12. Kneipp, W., Command of Multiple Actors Using Natural Language, B.Sc. (Honours) Thesis, Dept. of Computer Science, Univ. of New South Wales, Australia, 1985.
13. McDermott, J., R1: A Rule Based Configurer of Computer Systems, *Artificial Intelligence* 19(1):39–88 (1982).

14. Naish, L. and Thom, J., The MU Prolog Deductive Database, TR 83/10, Dept. of Computer Science, Univ. of Melbourne, Victoria, Australia, 1983.
15. Ramamohanarao, K., Lloyd, J. W., and Thom, J., Partial Match Retrieval Using Hashing and Descriptors, *ACM Trans. Database Systems* 8(4):552–576 (1983).
16. Ramamohanarao, K. and Shepherd, J., A Superimposed Codeword Indexing Scheme for Very Large Prolog Data Bases, presented at Third International Conference on Logic Programming, Imperial College of Science and Technology, London, 1986.
17. Roberts, C. S., Partial Match Retrieval via the Method of Superimposed Codes, *Proc. IEEE* 67(12):1624–1642 (1979).
18. Sabbatel, G. B. et al., Unification for Prolog Data Base Machine, in: *Proceedings of the Second International Logic Programming Conference*, Uppsala Univ., Sweden, 1984, pp. 207–217.
19. Salton, G., Advanced Information Retrieval Methods, in: *First Pan-Pacific Computer Conference*, Melbourne, Australia, 1985, pp. 118–133.
20. Sacks-Davis, R., Performance of a Multi-key Access Method Based on Descriptors and Superimposed Coding Techniques, *Inform. Systems* 10(4):391–403 (1985).
21. Sacks-Davis, R. and Ramamohanarao, K., A Two Level Superimposed Coding Scheme for Partial Match Retrieval, *Inform. Systems* 8(4):273–280 (1983).
22. Tick, E. and Warren, D. H. D., Towards a Pipelined Prolog Processor, in: *Proceedings of International Symposium on Logic Programming*, IEEE Computer Soc., 1984, pp. 29–40.
23. Warren, D. H. D., Implementing Prolog—Compiling Predicate Logic Programs, DAI Research Report Nos. 39, 40. Univ. of Edinburgh, Scotland, 1977.
24. Wise, M. W. and Powers, D. H. D., Indexing Prolog Clauses via Superimposed Code Words and Field Encoded Words, in: *Proceedings of International Symposium on Logic Programming*, IEEE Computer Soc., 1984, pp. 203–210.