Vectorization Techniques for Prolog

Yasusi Kanada, Keiji Kojima, and Masahiro Sugaya

Central Research Laboratory, Hitachi Ltd.
Kokubunji, Tokyo 185, Japan.

Abstract

Several techniques for running Prolog programs on pipelined vector processors, such as the Hitachi S-820 or the Cray-2, are developed. This paper presents an automatic program transformation (vectorization) method of Prolog, which enables a type of or-parallel execution of Prolog programs using vector operations. Performance is evaluated on the Hitachi S-810 using the Eight-Queens problem. Its vector execution speed is 4.5 MLIPS (18 ms). This is eight or nine times faster than scalar execution. This result confirms the effectiveness of vectorization techniques and applicability of vector processors to Prolog execution and to symbol processing applications.

1. Introduction

1.1 Prolog and Its High-Speed Execution

Prolog is a language with unification and automatic backtracking ('Prolog' is hereafter used to indicate logic programming languages in general). Unification is a very powerful pattern matching for data structures such as lists. Automatic backtracking makes programming of search problems easy. With these features, Prolog is quite suitable for symbol processing such as list processing, natural languuge processing, knowledge-base processing, and so on.

Although Prolog is a very powerful language, its execution speed is rather low. Thus, achieving high-speed execution is required. Many Prolog optimizing compilers in large scale general-purpose computers have been developed. However, higher speed requires parallel execution. There are two possible methods for parallel execution of Prolog. One is execution by parallel processors. The other is execution by pipelined vector processors such as the Hitachi S-820 or the Cray-2.

The former approach has been widely studied, for example, in the PIM (Parallel Inference Machine) [Ito 86, Ona 86] and Kabuwake method [Kum 86], etc.

The latter approach is proposed by Kanada [Kan 84, Kan 85] and attacked by him [Kan 87] using the Hitachi S-810 [Nag 84]. This approach is also being tried by Nilsson [Nil 86] also using the S-810, and by Tatsuguchi and Muraoka [Tat 87] using the IBM 3090 Vector Facility. However, their target is different from Kanada's (ours). Our target is to improve performance of Prolog with changing its semantics as little as possible. But their target is to improve performance of a logic programming language without backtracking, i.e., a so-called parallel logic programming language GHC [Ued 85] or its subset.

1.2 Difficulties in Improving Performance

Currently, most vector processors can only be used by Fortran or assembly language. Vectorizing compilers perform a program transformation called *vectorization* to generate the object program, which is a sequence of vector operations, from Fortran programs.

In the case of Fortran, vectorization techniques for most numerical programs have been established, and the program transformation is rather easy. This is because their main data structures are arrays that the vector processors are specialized to process, and because their control structures are simple DO loops in most cases.

However, the same techniques cannot be applied to Prolog, and drastic program transformations, such as introducing arrays and loops, are inevitable. There are three reasons for this.

The first reason is that it is difficult to accelerate data structure processing of Prolog, i.e., lists, functors, and so on. Prolog programs do not process arrays, but process (linked) lists which cannot be processed directly by vector operations.

The second reason is that Prolog programs do not have loops because Prolog lacks control structures like DO loops or GO TO statements. Instead, Prolog contains sophisticated control mechanisms like recursive call or automatic backtracking, whose vectorization techniques are undeveloped.

The third reason is that a unification is mapped to various operations case-by-case at machine level. These operations cannot be executed by a single uni-function vector instruction because unification is a bidirectional (multi-function) operation.

1.3 Contents of this Paper

This paper shows program transformation (vectorization) techniques of Prolog programs and the result of their performance evaluation. Section 2 gives a brief description of Prolog language and suggests possible vectorization strategies for Prolog programs. Section 3 outlines the vectorization methods. Section 4 gives the result of the performance evaluation of these methods. Section 5 gives the conclusion.

2. Prolog Language

Before explaining the vectorization of Prolog programs, a brief explanation of Prolog language, will be given. A detailed explanation is found in Clocksin and Mellish [Clo 81], for example.

2.1 Basics of Prolog

A Prolog program consists of procedures (subroutines). Each procedure consists of *clauses*. The form of a clause is as follows:

$p(a_1, a_2, \ldots, a_n) :- c_1, c_2, \ldots, c_m.$

The lefthand-side of :- (a *neck* symbol) is called the head of he clause, and the righthand-side is called the body. p is the name of the procedure, and $a_1, a_2, \ldots, a_n$ are the formal arguments. Thus a procedure consists of clauses that have the same name and the same number of formal arguments. The body consists of procedure calls $c_1, c_2, \ldots, c_m$. Some of them are calls of user-defined procedure and some of them are calls of system-defined ones.

When the procedure is called, one of the clauses whose head matches (can be unified with) the caller's pattern is selected and the body of the clause is executed. If no head matches, the execution is said to have *failed*. If one of the procedure calls in the body has failed, execution of the clause has also failed. Otherwise, it is said to be *succeeded*. If execution of a clause fails, another clause whose head matches is selected and executed. This action is called *automatic backtracking*.

Two examples will be shown in the next two sections. One of them is a deterministic procedure, which means that the procedure has a single solution, or always only one clause of the procedure succeed. The other is a nondeterministic procedure, which means that it has multiple solutions, or more than one clause of the procedure may succeed. They are included because the execution and the vectorization methods of nondeterministic procedures are different from those of deterministic procedures. Mode declaration, which plays an important role on checking determinacy, is explained in Section 2.4.

2.2 A Deterministic Example

The following example is the well-known append procedure, which concatenates two lists.

```
append([], X, X).                                    ...... (2.1)
append([H | R], Y, [H | R1]) :- append(R, Y, R1).    ...... (2.2)
```

For example, the execution of the following question results in Z = [a, b, c, d, e]:

```
?- append([a, b, c], [d, e], Z).                     ...... (2.3)
```

This question means "what is the concatenation of list [a, b, c], and list [d, e]." The answer is list [a, b, c, d, e] (A list is bracketed in Prolog). In this question, [a, b, c] and [d, e] are constant lists (all the elements are atomic symbols) and Z is a variable. As far as the first argument is used as input, only one clause always success during the execution of the above procedure. Thus the append procedure is deterministic.

The append procedure has no loops, but has a tail recursive call. This means that the righthand-side of the second clause is a procedure call of append itself. The recursive call appears at the end of the procedure, so it is called a tail recursive call. A loop cannot be expressed by Prolog, but a procedure with a tail recursive call can be transformed into a loop in procedural languages, such as Fortran or Pascal. Such a loop may be vectorized. This fact suggests a possible strategy for vectorization.

2.3 A Nondeterministic Example

The append procedure can also be used to split a list.

```
?- append(X, Y, [a, b]).                             ...... (2.4)
```

Question 2.3 results in a single answer, but question 2.4 results in multiple answers. The question means "what concatenated with what gives us [a, b]." In this case, the question has three answers,

```
(1) X = [],      Y = [a, b],
(2) X = [a],     Y = [b],
(3) X = [a, b],  Y = [].
```

where [] means an empty list. The question has multiple answers because both clauses of append have succeeded. Such a question is said to be nondeterministic.* Nondeterminacy is one of the important characteristics of Prolog.

The above example suggests a possible vectorization strategy, that is, for computing different solutions in a pipelined manner. Thus, there is the possibility of applying vector operations by changing backtracking to looping.

2.4 Mode Declarations

Although the above append procedure can be used for both concatenating and splitting a list, such a multi-function procedure may be less efficient than a specialized, uni-function one. A language feature called *mode declaration*, used to generate more efficient code, is supported by many Prolog systems.

An example of a mode declaration for append is as follows:

```
mode append( +, +, - ).                          ...... (2.5)
```

This mode declaration means the first and the second argument of the append procedure is input, and the third one is output.

If mode declaration 2.5 is supplied for the append procedure, question 2.3 will be executed faster, but question 2.4 might not be answered correctly because it is a wrong usage. On the other hand, if the following mode declaration is supplied, question 2.4 will be executed faster, but question 2.3 might not be answered correctly.

```
mode append( -, -, + ).                          ...... (2.6)
```

3. Vectorization Methods of Prolog Programs

3.1 And-Vectorization and Or-Vectorization

Two possible vectorization strategies were suggested in the last section.

(1) pipelined execution of "tail recursive calls", and
(2) pipelined execution of "backtracking".

For parallel processing of logic programming languages such as Prolog, parallelism is roughly classified into and-parallelism and or-parallelism [Con 81]. And-parallelism is the parallelism between processes that are part of computation getting a single solution, and or-parallelism is the parallelism between computation getting multiple solutions (Note that the meanings of these words are more general than the usual definitions). The method based on (1) is a type of and-parallelization, so this program transformation is called *and-vectorization*. The method based on (2) is a type of or-parallelization, so it is called *or-vectorization*. The rest of this paper concentrates on or-vectorization. The and-vectorization method is currently being developed and its details will be described in a future paper.

3.2 Basic Ideas of Or-Vectorization

The three problems listed in Section 1.2 must be solved in order to achieve vectorization. Their details in the case of or-vectorization are explained below.

(1) Data structure problem

The original Prolog program generates multiple solutions, which are generated one by one. A solution never coexists with another solution in a sequential execution. However, they must coexist in an array, if they are to be processed by a vector processor. So it is necessary to generate the code of accumulating solutions in different alternatives into an array.

(2) Control structure problem

A program with automatic backtracking must be converted to a collection of very small loops without backtracking. Each loop is executed by a single vector operation.

(3) Unification problem

It is necessary to perform unifications in vector operations to realize drastic improvements in performance. In the execution of a program like the Eight-Queens problem which has a large amount of or-parallelism, many unifications are done. Thus improvement is possible. The following are subproblems in unification:

(3A) Because list is the most important data structure, element-wise unification of lists contained in arrays should be performed by vector operations at high-speed.

(3B) Compilation of Prolog patterns into uni-function vector unifiers such as list composer, list decomposer, etc., are necessary.

3A is mainly an execution mechanism problem, and 3B is a vectorization techniques problem.

The following methods were used to solve these problems.

(1) Accumulation of solutions and introducing loops

In this method, values of a logical variable in different alternatives (execution paths) are accumulated in an array at the end of a nondeterministic procedure execution.

The execution process of a nondeterministic procedure is as follows. All the clauses of the procedure are executed in order before exiting the procedure. On the other hand, only one clause is executed before exiting from it in the original semantics of Prolog (if the clause succeeded).

Each transformed clause inputs and outputs vectors that contain values of original arguments. The output vectors are merged at the end of the procedure execution. For example, the outline of the vector process of question 2.4 is shown in figure 1. In this case, the length of the input vectors is one. The three solutions are obtained sequentially, and accumulated into arrays. So the length of the output vectors is three. The elements of the output vectors should be in depth-

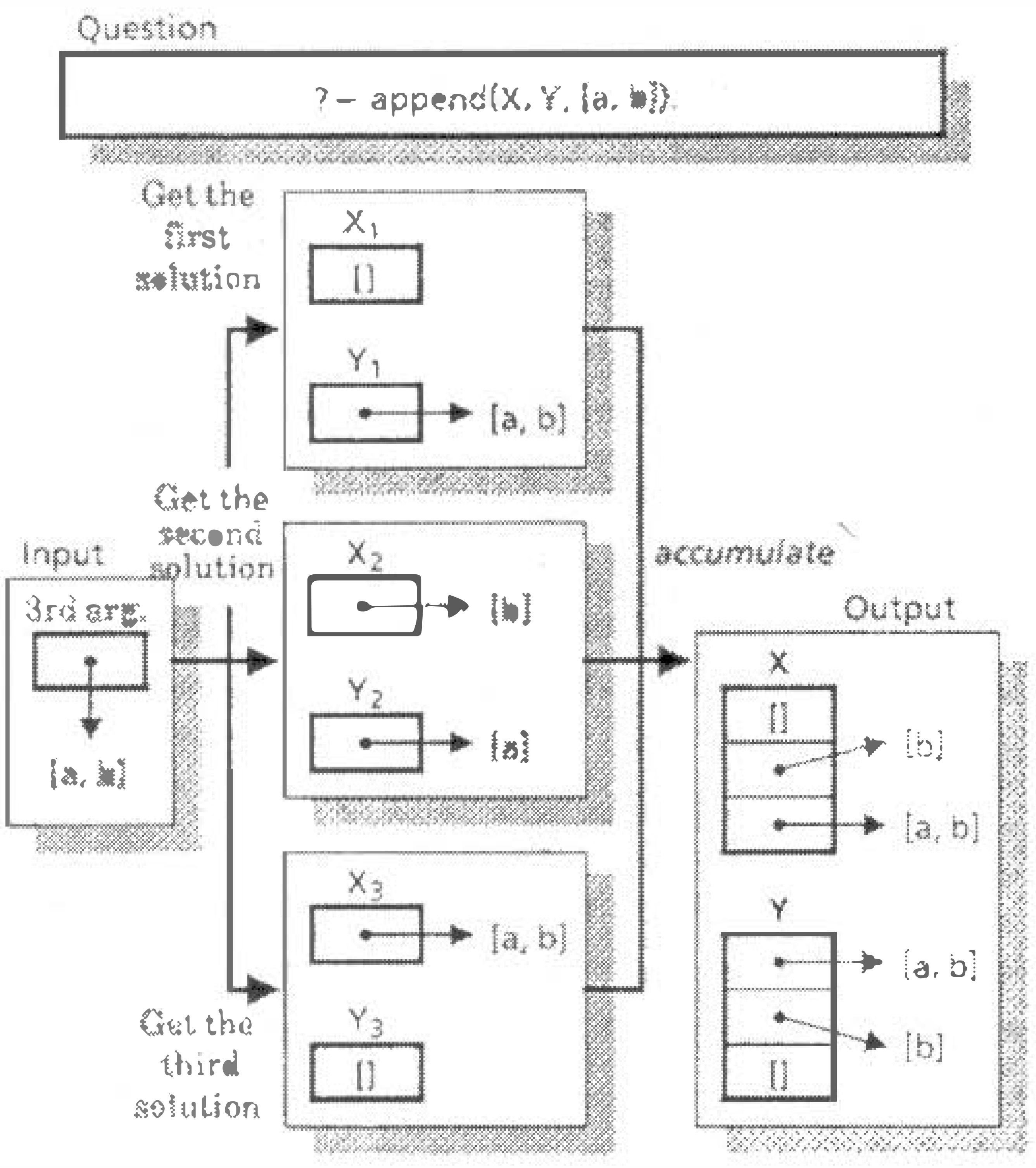


Figure 1. The outline of the vector processing of the nondeterministic 'append' procedure

first order, which is necessary for keeping the order of the final solutions the same as in sequential execution.

The above accumulation is not done in or-parallel, but each following operation on the accumulated data can be done in or-parallel by vector operations. It is better to do the accumulation by vector operations, because the length of the input vector may be greater than one.

This program transformation has some similarity to the transformation of nondeterministic Prolog programs into GHC by Ueda [Ueda 87] and Tamaki [Tam 87].

(2) Vector-element-wise operations on lists

To process arrays that contain linked lists, it is necessary to perform vector-element-wise list operations with a vector processor. That is, the same operations on multiple lists, which are the elements of an array, must be performed at once. List vector operations [Kam 83] (or indexed vector load/store operations) can be used for this purpose. (It is difficult to implement Prolog on a vector processor without list vector operations).

The basic scalar list operations and their execution methods using vector operations are shown in figure 2. The notation #(A, B,..., Z) is used for a vector containing A, B, ..., Z. In the case of decomposition, the heads and the tails of the vector elements are obtained using two indexed vector load operations (a kind of list vector operation) and are stored in

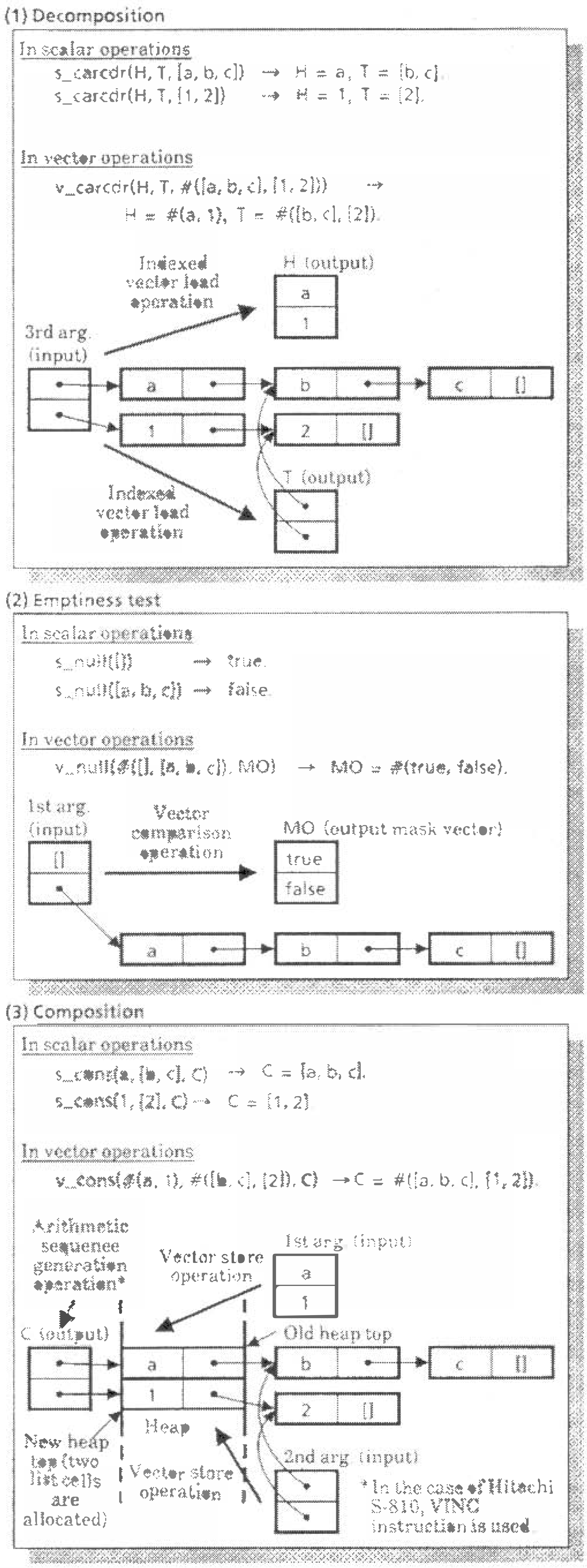

(1) Decomposition
In scalar operations
s_carcdr(H, T, [a, b, c]) → H = a, T = [b, c].
s_carcdr(H, T, [1, 2]) → H = 1, T = [2].
In vector operations
v_carcdr(H, T, #([a, b, c], [1, 2])) →
H = #(a, 1), T = #([b, c], [2]).
Indexed vector load operation
H (output)
3rd arg. (input)
T (output)
Indexed vector load operation
(2) Emptiness test
In scalar operations
s_null([]) → true.
s_null([a, b, c]) → false.
In vector operations
v_null(#([], [a, b, c]), MO) → MO = #(true, false).
1st arg. (input)
Vector comparison operation
MO (output mask vector)
true
false
(3) Composition
In scalar operations
s_cons(a, [b, c], C) → C = [a, b, c].
s_cons(1, [2], C) → C = [1, 2].
In vector operations
v_cons(#(a, 1), #([b, c], [2]), C) → C = #([a, b, c], [1, 2]).
Arithmetic sequence generation operation*
Vector store operation
1st arg. (input)
C (output)
Old heap top
Heap
New heap top (two list cells are allocated)
Vector store operation
2nd arg. (input)
* In the case of Hitachi S-810, VINC instruction is used.

Figure 2. Basic list operations and their vector counterparts

output vectors H and T. In the case of the emptiness test, a vector comparison operation is used and the result is expressed by boolean vector (called a mask vector) MO. In the case of composition, the cons cells are allocated on the heap, and the value of the first and the second arguments are stored in them. The addresses of the cons cells is calculated by an arithmetic sequence generation operation and stored in output vector C.

There is also the problem of conditional control. In the case that some unifications on vector elements fail and others succeed, this failure or success must be stored in memory. The operations following the unifications are controlled by the stored data. There are three conditional control methods. They are explained in Section 3.3.

(3) Compilation to uni-function unifiers

In this method, unifications in a source program are compiled into calls of uni-function unifiers, using mode information of the procedure's arguments and variables. The word *mode information* refers to information that shows whether the arguments or variables are used only for input, only for output, or for both. Mode information may be supplied with mode declarations by programmers. Although most procedures are always used in the same mode, it is troublesome for programmers to write mode declarations for all the procedures. Thus automatic mode analysis is necessary. But currently this system is fully dependent on mode declarations.

3.3 Three Conditional Control Methods

In vector operations of vector processors, conditional control is performed mainly by the following three kinds of instructions [Kam 83]:

(1) Masked operation instructions,
(2) List vector instructions,
(3) Gather/scatter instructions.

The methods that follows are used for the conditional control of vectorized Prolog programs. Each of them uses the instructions which shares the same number. Input and output of the pipelined execution of questions with these methods are illustrated in figure 3.

(1) Masked operation method

In execution by this method, each array may contain dead elements (all arrays have the same number of elements). The dead elements are shaded in figure 3. "Live-ness" of array elements is shown by a *mask vector*. The mask vector is a boolean vector. The cost of masked operations is very low in vector processors. However, if the percentage of dead elements

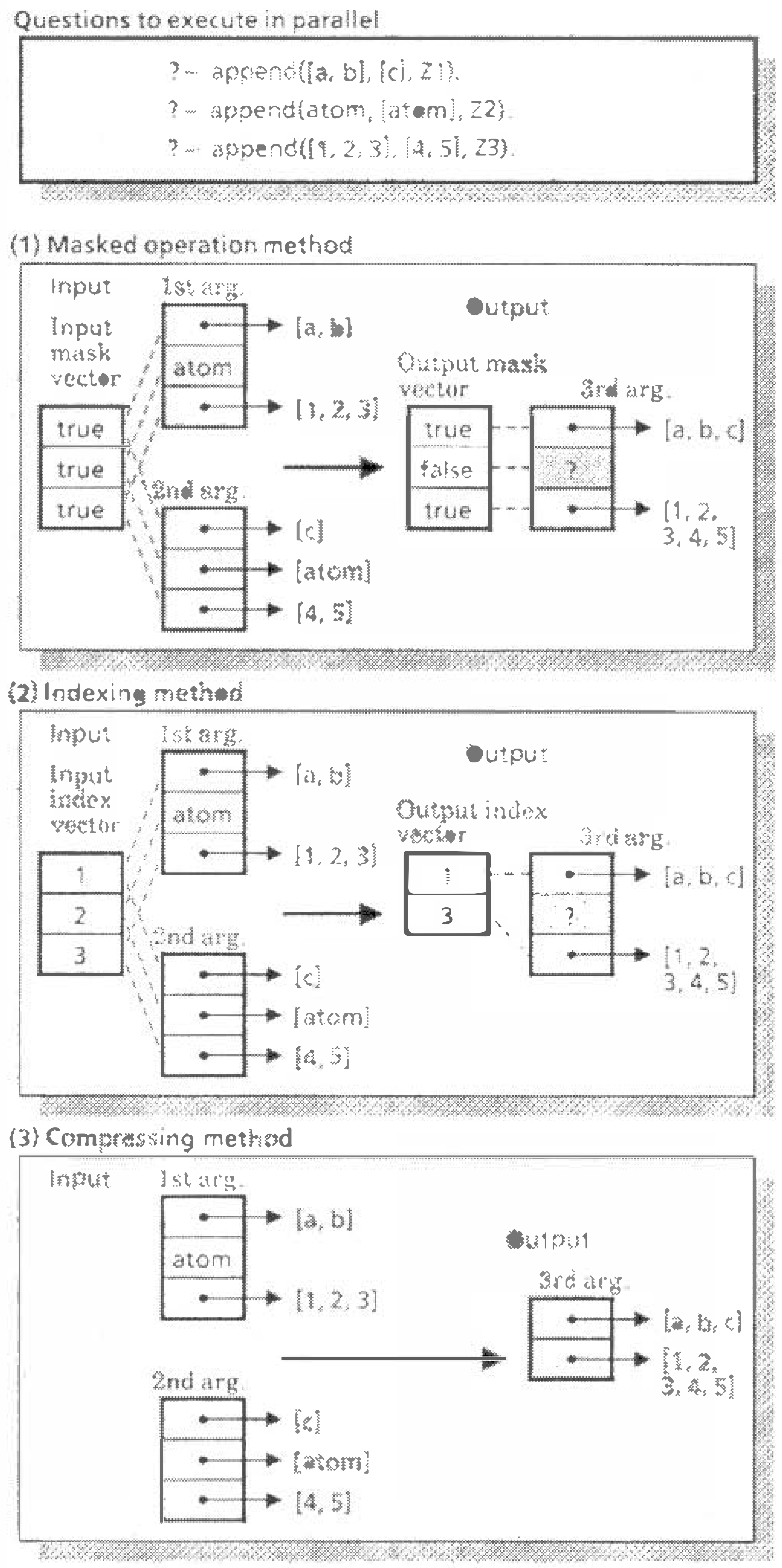


Figure 3. The three conditional control methods

is large, this method is inefficient because dead element access overhead exists.

(2) Indexing method

Similar to the masked operation method, each array may contain dead elements. Indices (or displacements) of live elements are stored in an *index vector*, whose number of elements is the same or less than that in data arrays. Although no dead element is accessed, list vector access overhead, which is expensive in vector processors, always exists in this method.

(3) Compressing method.

In this method, each array consists only of live elements. No control vector, such as a mask vector or an index vector, is used. No dead element is accessed, so no access overhead exist. However, if some elements in vectors become "dead", all the vectors should be compressed to keep the correspondence of vector elements. So this method is inefficient in many cases.

3.4 Process of Compilation

This section gives a brief description of the compilation process and shows the position of the vectorization. Sections 3.5 and 3.6 describe the details of or-vectorization methods. Section 3.5 describes those for deterministic procedures and section 3.6 describes those for nondeterministic procedures .

Because drastic program transformation is necessary, Prolog execution in vector processors should be based not on an interpreter, but on a compiler. The compilation process used in this paper is shown in figure 4. In this paper's execution method,

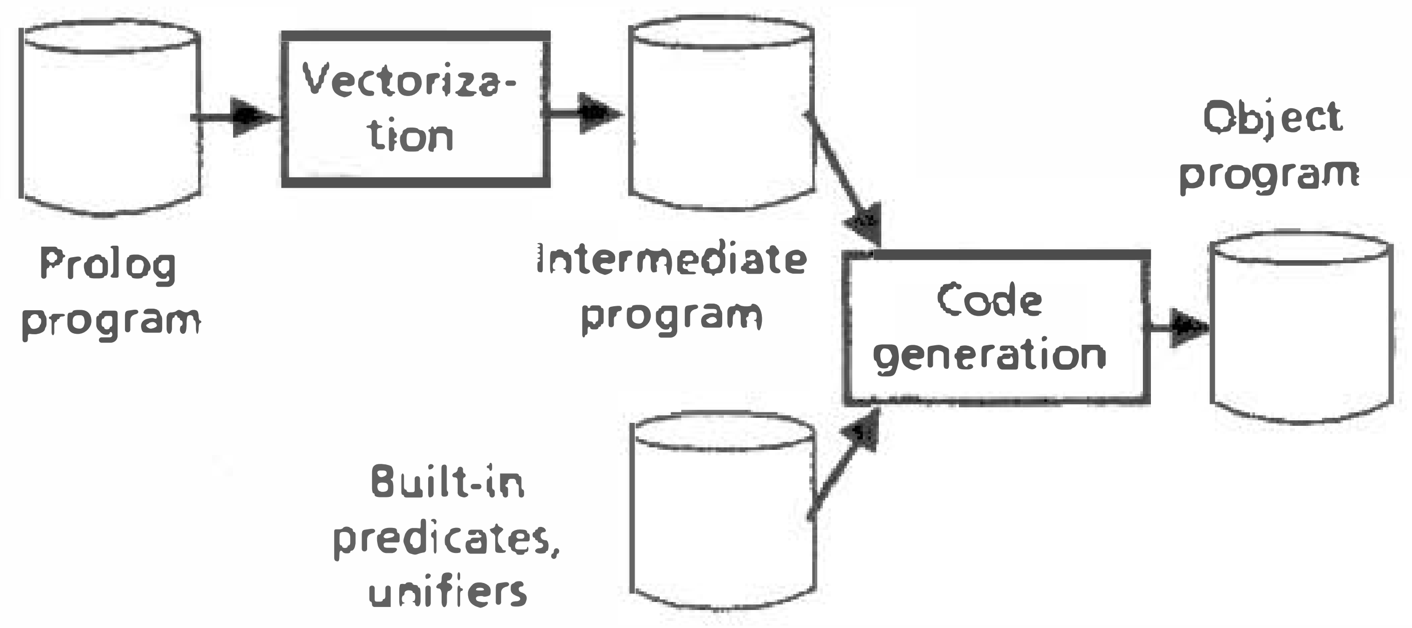


Figure 4. Compilation process of Prolog for vector processors

the compilation process of Prolog programs consists of two phases. The first phase is called *vectorization*, and the second phase is called *code generation*.

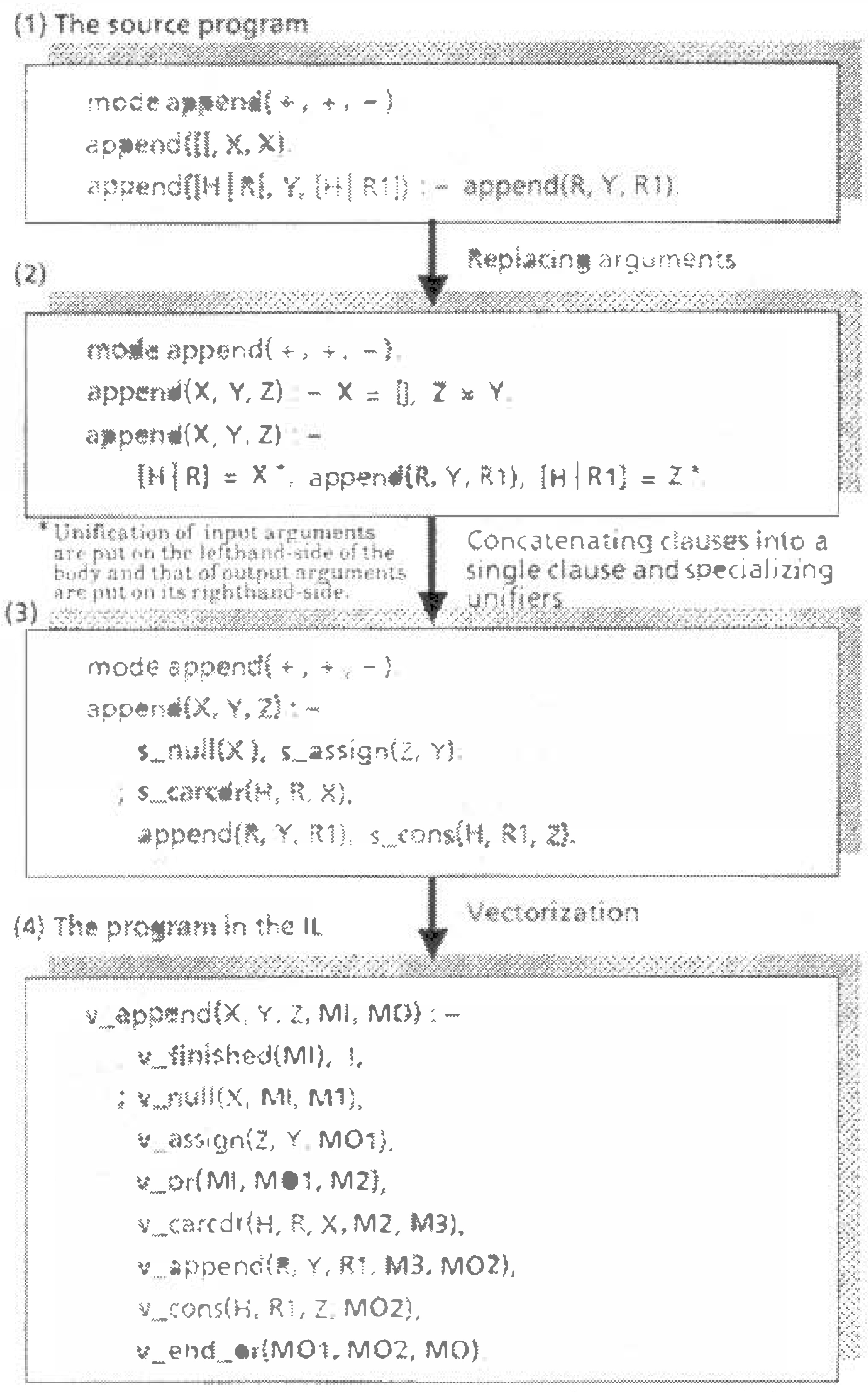
Vectorization is a program transformation. The resulting program is expressed in an intermediate language (IL) which may be a high-level language. A Prolog-like language is used as IL in this paper, for the sake of simplicity. The IL is very similar to Prolog but it contains arrays.

Code generation is a simple process in most cases. It is similar to the compilation process of the Fortran vectorizing compiler.

3.5 Or-Vectorization of Deterministic Procedures

The vectorized form and the outline of the vectorization process of the append procedure with mode declaration 2.5, i.e., deterministic append, is shown in figure 5.

The first line of the source program is the mode declaration. In the current implementation, it is indispensable for vectorization.


(1) The source program

```
mode append( + , + , - )
append([], X, X).
append([H | R], Y, [H | R1]) :- append(R, Y, R1).
```

(2)

```
mode append( + , + , - ).
append(X, Y, Z) :- X = [], Z = Y.
append(X, Y, Z) :-
    [H | R] = X *, append(R, Y, R1), [H | R1] = Z *.
```

* Unification of input arguments are put on the lefthand-side of the body and that of output arguments are put on its righthand-side.

Concatenating clauses into a single clause and specializing unifiers

(3)

```
mode append( + , + , - ).
append(X, Y, Z) :-
    s_null(X ), s_assign(Z, Y).
  ; s_carcdr(H, R, X),
    append(R, Y, R1), s_cons(H, R1, Z).
```

(4) The program in the IL

```
v_append(X, Y, Z, MI, MO) :-
    v_finished(MI), !,
  ; v_null(X, MI, M1),
    v_assign(Z, Y, MO1),
    v_or(MI, MO1, M2),
    v_carcdr(H, R, X, M2, M3),
    v_append(R, Y, R1, M3, MO2),
    v_cons(H, R1, Z, MO2),
    v_end_or(MO1, MO2, MO)
```

Figure 5. Vectorization process of the deterministic 'append' procedure

However, it will be unnecessary in most cases, if interprocedural automatic mode analysis is done.

The process of program transformation can be divided into three steps. The real process is more complicated, but it is simplified here. First, the arguments in the source program are replaced by variables. Second, the clauses are concatenated into a single clause using ';' (the built-in *or* predicate), and unifiers are specialized. Finally, the vectorization is performed. The vectorization shown in figure 5 is based on the masked operation method. Other methods are explained in the appendix. In the IL in figure 5(4), the arguments of v_append are vectors. The first three formal arguments (i.e., X,* Y and Z) correspond to the arguments of the original append. Fourth argument MI is the input mask vector, and fifth one MO is the output mask vector.

An array with elements e_1, e_2, ..., e_n will be described as #(e_1, e_2, ..., e_n). Then, the execution of the following question results in

Z = #([a, b, c], [1, 2, 3, 4, 5], ?) and MO = #(true, true, false), where ? is an arbitrary value.

```
?- v_append(#([a, b], [1, 2, 3], atom), #([c], [4, 5], [atom]),
            Z, #(true, true, true), MO).                      ...... (3.1)
```

The vectorized append, named v_append here, computes the element-wise list concatenation of two vectors. The first two element processings are succeeded, so the values of the first two elements of the output mask vector become true. The third element processing is failed, so the value of the third element of the output mask vector is false. The correspondence between the source program expressions and the vectoriz d program ones is shown in table 1.

Table 1. The correspondence between the source program and the vectorized one for the deterministic 'append' procedure

Source Form	Vectorized Form	Meaning
-*	v_finished(MI)	Stop recursion.
[]	v_null(X, MI, MO1)	Test emptiness of lists†.
X, X	v_assign(Z, Y, MO1)	Assign the 2nd argument to the 1st.
The two clauses	v_or(MI, MO1, M2), and v_end_or(MO1, MO2, MO)	Merge the result of the two clauses.
[H | R]	v_carcdr(H, R, X, M2, M3)	Decompose the lists†.
append(R, Y, R1)	v_append(R, Y, R1, M3, MO2)	Call recursively.
[H | R1]	v_cons(X, R1, Z, MO2)	Compose lists†.

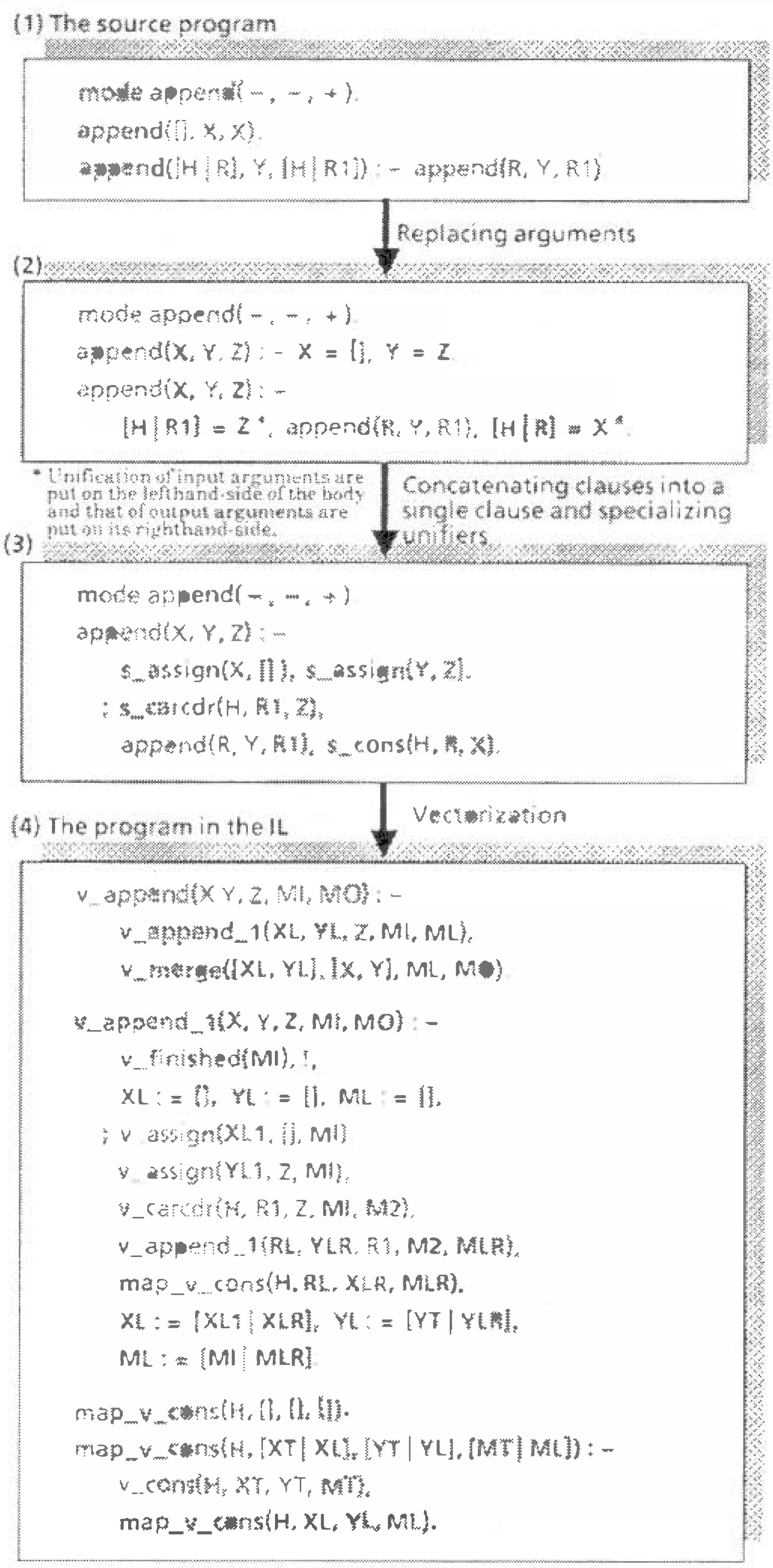
* There is no counterpart in the source program.
† The elements of the arguments are lists.

The vectorized program contains three procedure calls which have no counterparts in the source program, i.e., v_finished, v_or and v_end_or. The reason why procedure call v_finished(MI) is inserted is that the recursion of v_append never stops without it (The v_finished tests whether there is true in the argument mask vector). The v_or procedure prepares mask vector M2 for the parts following it, which correspond to the second clause of the source program. Then, the v_end_or procedure makes output mask vector MO.

3.6 Or-Vectorization of Nondeterministic Procedures

The vectorized form and the outline of the vectorization process of the append procedure with mode declaration 2.6, i.e., nondeterministic append is shown in figure 6. The vectorization process is also divided into three steps, i.e., replacing arguments, concatenating clauses into a single clause and specializing unifiers, and vectorization. The first two steps are done the same way as the deterministic case. However, the output is different because the mode declaration is different.

The vectorization shown in figure 6 is based also on the masked operation method. The first three formal arguments (i.e.,


(1) The source program
Replacing arguments
(2)
Concatenating clauses into a single clause and specializing unifiers
(3)
Vectorization
(4) The program in the IL

```
mode append(-, -, +).
append([], X, X).
append([H | R], Y, [H | R1]) :- append(R, Y, R1).
```

```
mode append(-, -, +).
append(X, Y, Z) :- X = [], Y = Z.
append(X, Y, Z) :-
    [H | R1] = Z *, append(R, Y, R1), [H | R] = X *.
```

* Unification of input arguments are put on the lefthand-side of the body and that of output arguments are put on its righthand-side.

```
mode append(-, -, +)
append(X, Y, Z) :-
    s_assign(X, []), s_assign(Y, Z).
  ; s_carcdr(H, R1, Z),
    append(R, Y, R1), s_cons(H, R, X).
```

```
v_append(X Y, Z, MI, MO) :-
    v_append_1(XL, YL, Z, MI, ML),
    v_merge([XL, YL], [X, Y], ML, MO).

v_append_1(X, Y, Z, MI, MO) :-
    v_finished(MI), !,
    XL := [], YL := [], ML := [].
  ; v_assign(XL1, [], MI)
    v_assign(YL1, Z, MI),
    v_carcdr(H, R1, Z, MI, M2),
    v_append_1(RL, YLR, R1, M2, MLR),
    map_v_cons(H, RL, XLR, MLR).
    XL := [XL1 | XLR], YL := [YT | YLR],
    ML := [MI | MLR].

map_v_cons(H, [], [], []).
map_v_cons(H, [XT | XL], [YT | YL], [MT | ML]) :-
    v_cons(H, XT, YT, MT),
    map_v_cons(H, XL, YL, ML).
```

Figure 6. Vectorization process of the nondeterministic 'append' procedure

X, Y and Z) of v_append correspond to the arguments of the original append. The fourth and the fifth formal arguments (MI and MO) are the input and output mask vectors. The vectorized program consists of procedures v_append, v_append_1, and map_v_cons. Their meaning is explained later.

The execution of the question 3.2 results in the following values of X and Y.

```
?- append(X, Y, #([a, b], [1])).                      ...... (3.2)
```

```
X = #([],     [], [a], [1], [a, b]),
Y = #([a, b], [1], [b], [],  []).
```

The vectorized program consists of three procedures, i.e., v_append, v_append_1 and map_v_cons. The relation between the source program expressions and the vectorized program ones is shown in table 2.

Table 2. The correspondence between the source program and the vectorized one for the nondeterministic 'append' procedure

Source Form	Vectorized Form	Meaning
-*	v_merge([XL, YL], [X, Y], ML, MO)	Accumulate the solutions.
-*	v_finished(MI)	Stop recursion
-*	XL := [], YL := [], ML := []	Make empty multi-vectors.
[]	v_assign(XL1, [], MI)	Make XL1 a vector of empty lists†.
X, X	v_assign(YL1, Z, MI)	Assign the 2nd argument to the 1st†.
[H | R1]	v_carcdr(H, R1, M2, M3)	Decompose the lists†.
append(R, Y, R1)	v_append_1(RL, YLR, R1, M2, MLR)	Call recursively.
[H | R]	map_v_cons(H, RL, XLR, MLR)	Compose lists†.
-*	XL := [XL1 | XLR], YL := [YL1 | YLR], ML := [MI | MLR]	Add an element to each multi-vector.

* There is no counterpart in the source program.
† The elements of the arguments are lists.

v_append is the main procedure. It calls v_append_1 to get solutions in the form of chained vectors, which we call multi-vectors, and calls v_merge to accumulate the values of each variable into a single vector. The computation process of question 3.2 is shown in figure 7. In this case, element vectors of multi-vectors XL and YL are merged into vectors X and Y respectively. ML is a multi-vector whose elements are mask vectors. ML shows the "live-ness" of XL and YL. v_merge outputs a vector with no dead elements, so all the elements of output mask vector MO is true.

The expressions such as XL := [], XL := [XL1 | XLR] in the vectorized program are used for making a multi-vector. The former is used for making an empty multi-vector (a multi-vector that has no elements), and the latter is used for adding an element vector to a multi-vector.

The recursive call of v_append_1 yields multi-vectors RL, YLR and MLR. The call of map_v_cons inputs H and RL, computes the vector-element-wise list composition of H and each elements of multi-vector RL, and outputs multi-vector XLR whose elements are the results of the composition.

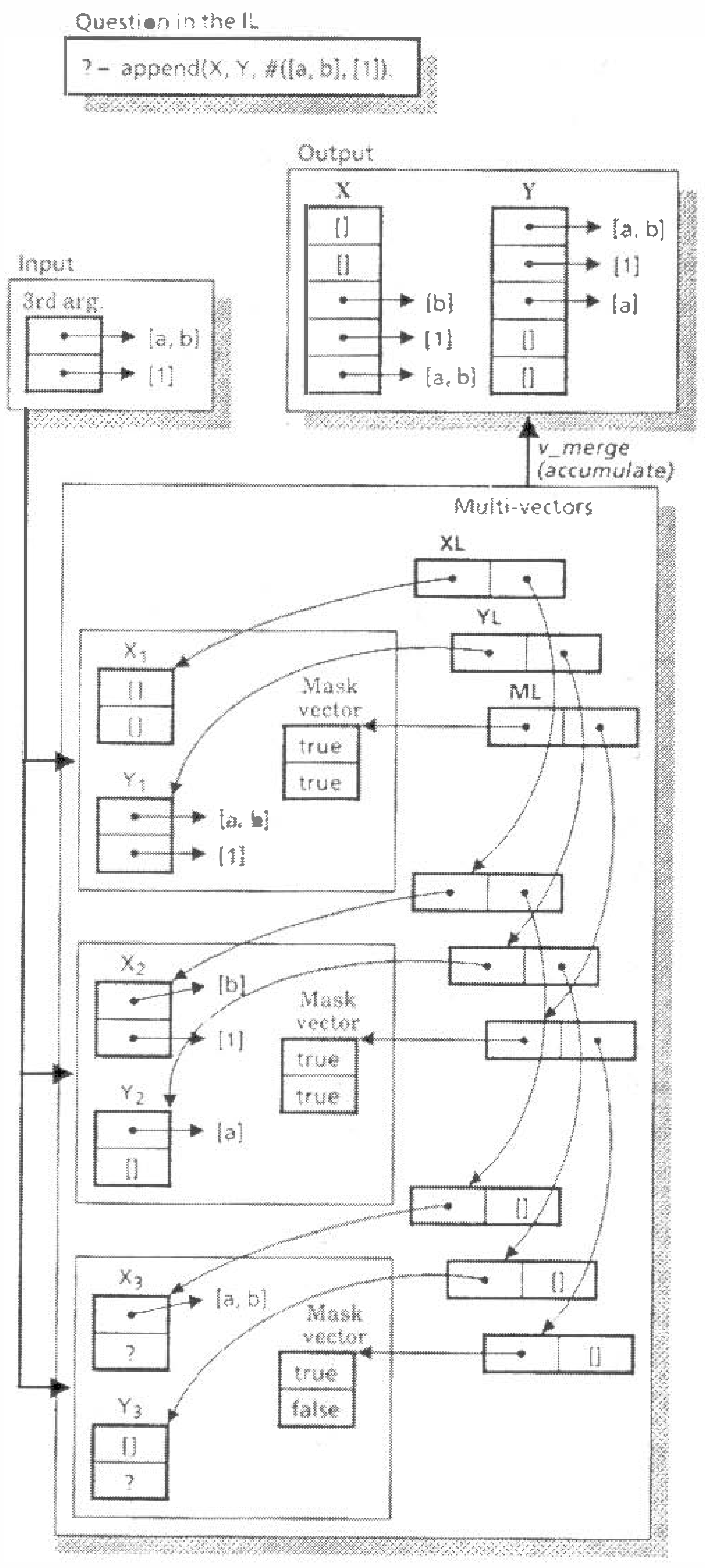


Figure 7. The vector computation process of the nondeterministic 'append' procedure

In the current method, backtracking is completely eliminated, so the number of vector elements may explode during the execution of a program like the N-Queens problem (a generalization of the Eight-Queens problem) when N is large. If the or-vectorization method and backtracking are combined, this explosion can be avoided. A schema to combine them, called the *parallel backtracking schema* [Kan 84, Kan 85], has been designed.

4. A Performance Evaluation

Before developing a Prolog compiler with an automatic vectorizer, vector processing performance of Prolog programs was manually evaluated. The program of the Eight-Queens problem was manually vectorized by the masked operation, indexing and compressing methods into IL, and then translated into Fortran and Pascal programs. The program portion that can be vectorized by the Fortran compiler is written in Fortran, and program portion that uses recursion is written in Pascal. The source program and the parts of the vectorized program are shown in the appendix. A garbage collector has not been implemented.

Vector and scalar execution performance is shown in table 3.

Table 3. Performance of the Eight-Queens program on the Hitachi S-810

Method of conditional control	S-810 vector execution		S-810 scalar execution		Accelera-tion rate
	MLIPS	time (ms)	MLIPS	time (ms)	
Masked operation	4.5	18	0.48	167	9.3
Indexing	4.5	18	0.57	140	7.8
Compressing	4.2	19	0.50	160	8.4
(array version)	–	9	–	79*	8.8*

* The scalar execution time of the array version is that of a program optimized for scalar execution. The program is different from that of the vector execution time.

The scalar execution time is the time when the vectorization feature of the Fortran compiler is suppressed. The execution time of the Eight-Queens program using arrays instead of lists [Kan 84] (array version) is also included in table 3, for comparison. The percentage of the total execution time required by each type of operation is shown in figure 8.

The important points are as follows.

(1) Vector execution time is 18 to 19 ms. It means that inference speed is 4.2 to 4.5 MLIPS (million logical inferences per second). No significant difference in performances exists between masked operation, indexing and compressing methods in the Eight-Queens case. The vector processes are eight or nine times faster than the scalar processes.

(2) The percentage of the total execution time required by the list emptiness test, list composition and list decomposition is high in all three methods (about 40% in vector processing).

(3) The accelerating rate of list composition is low. This low rate shows that the throughput between the main storage and the vector processing unit is insufficient in the list composition case.

(4) The percentage of the total execution time required by compressing vectors and merging (accumulating) vectors is about 6% in the masked operation and indexing methods, but it is about 25% in the compressing method. The overhead of the latter cannot be ignored.

5. Conclusion

Vectorization methods for Prolog, called *or-vectorization methods*, which enables a type of or-parallel execution of Prolog on vector processors, have been developed. There are three conditional control methods of vector processing, namely, the masked operation, indexing and compressing methods. These three methods have been manually evaluated. No significant time difference between these three methods has been found in solving the Eight-Queens problem. They will be useful for solving search problems or for other symbol processing applications in the future because of their high processing speeds.

Acknowledgement

The authors wish to thank Dr. Sakae Takahashi and Seiichi Yoshizumi of Hitachi Ltd. for their continuing support of our research.

References

[Clo 81] Clocksin, W. F., and Mellish, C. S., "Programming in Prolog", Springer-Verlag, New York, 1981.

[Con 81] Conery, J. S., and Killer, D. F., "Parallel Interpretation of Logic Programs", *ACM 1981 Conference on Functional Programming Languages and Computer Architecture*, pp.163-170.

[Ito 86] Ito, N., et. al., "The Architecture and Preliminary Evaluation Results of the Experimental Parallel Inference Machine PIM-D", *13th Annual International Symposium on Computer Architecture*, pp.149-156, 1986.

[Kam 83] Kamiya, S., Isobe, F., Takashima, H., and Takiuchi, M., "Practical Vectorization Techniques for the "FACOM VP"", *Information Processing 83*, pp.389-394, Elsevier Science Publishers B. V., 1983.

[Kan 84] Kanada, Y., "A Schema for Solving N-Queens Problem by a Vector Processor: Parallel Backtracking Schema", *29th National Conference of Japan Society of Information Processing*, pp.1251-1252, 1984 (in Japanese).

[Kan 85] Kanada, Y., "Improving Prolog Performance using Supercomputer", Proceedings of 26th Programming Symposium, pp.47-56, 1985 (in Japanese).

[Kan 87] Kanada, Y., "Or-parallel Vector Processing Methods for N-Queens: Towards High-Speed Execution of Logic Programs on Vector Processors", *Technical Group on Programming Language, Japan Society of Information Processing*, 87-PL-12, 1987 (in Japanese).

[Kum 86] Kumon, K., et. al., "KABU-WAKE: A New Parallel Inference Method and Its Evaluation", *COMPCON Spring 86*, pp.168-172, 1986.

[Nag84] Nagashima, S., et. al., "Design Consideration for High-Speed Vector Processor: S-810", *Proceedings of IEEE International Conference on Computer Design*, pp.238-242, 1984.

[Nil 86] Nilsson, M., "– FLENG Prolog – The Language which turns Supercomputers into Parallel Prolog Machines", *Logic Programming '86* (in Japan), pp.209-216, 1986.

[Ona 86] Onai, R., Shimizu, H., Masuda, K., Matsumoto, A., and Aso, M., "Architecture and Evaluation of a Reduction-Based Parallel Inference Machine: PIM-R", *Logic Programming '85*, Lecture Notes in Computer Science, No.221, pp.1-12, Springer-Verlag, 1986.

[Tam 87] Tamaki, H., "Stream-Based Compilation of Ground I/O Prolog into Committed-choice Languages", *4th International Conference on Logic Programming*, pp.376-393, Melborne, 1987.

[Tat87] Tatsuguchi, K., and Muraoka, Y., "Parallel Logic Programming Language Interpreters on Supercomputers", *Technical Group on Programming Language*, Japan Society of Information Processing, 87-PL-14,1987 (in Japanese).

[Ueda 85] Ueda, K., "Guarded Horn Clauses", *ICOT Technical Report*, TR-103, Institute for New Generation Computer Technology, 1985.

[Ueda87] Ueda, K., "Making Exhaustive Search Programs Deterministic", *3rd International Conference on Logic Programming*, Lecture Notes in Computer Science, No.225, pp.270-282, Springer-Verlag, 1987.

Appendix 1. The Vectorized 'Append' Procedure

Though the process and the results of the vectorization of the append procedure by the masked operation method are shown in Section 3.5, those by the indexing and compressing methods are not described there. Only their usage is shown below.

```
?- x_append(#([a, b], [1, 2, 3], atom), #([c], [4, 5], [atom]),
            Z, #(1, 2,3), XO).                              ...... (A.1)
```

It returns the same result as question 3.1, i.e., Z = #([a, b, c], [1,2, 3,4, 5], ?). The fourth argument #(1, 2, 3) is the input index vector which shows the first and the second element of the input arguments are live. The fifth argument XO is the output index vector, whose value becomes #(1, 2) after the execution.

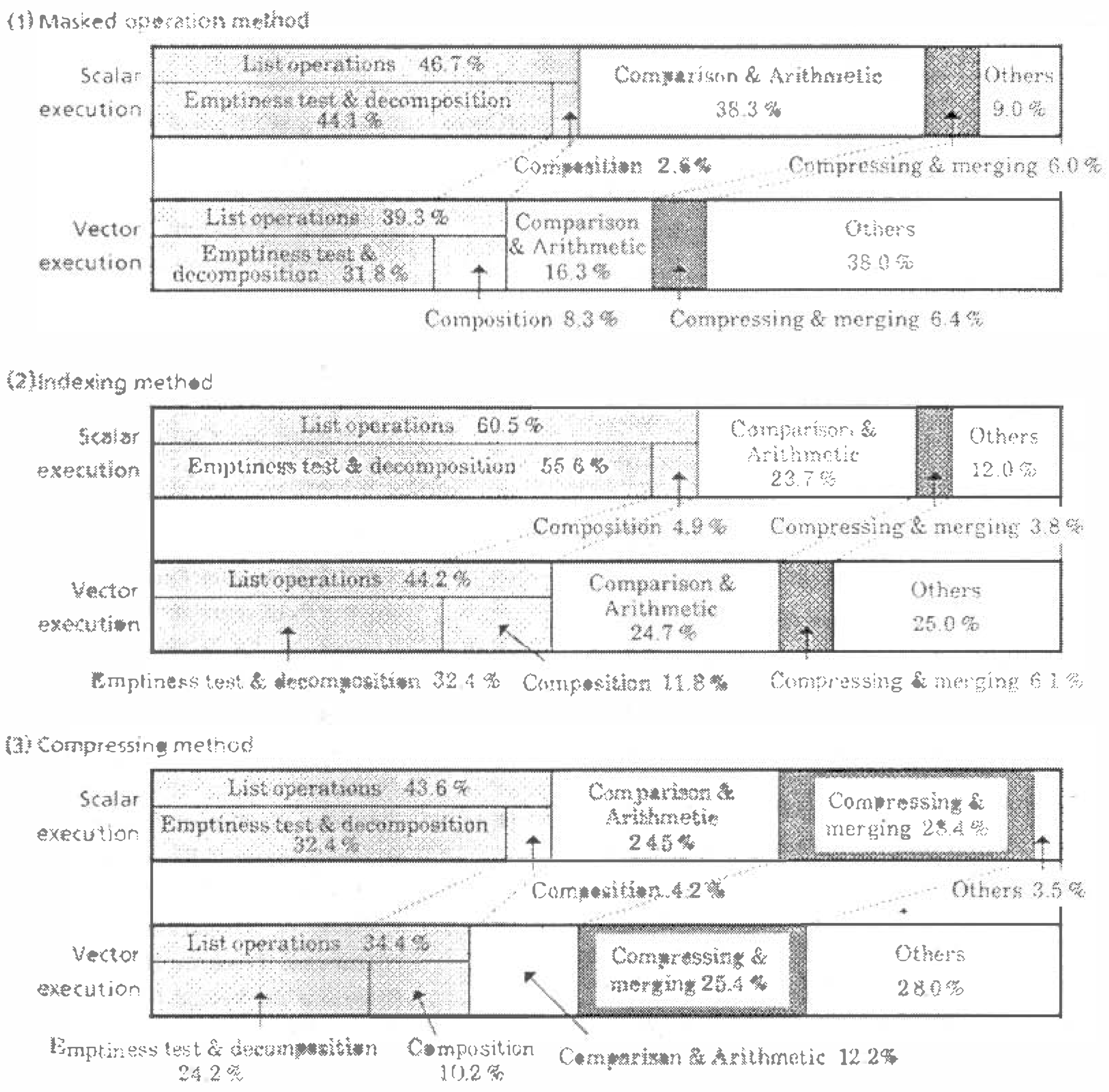


Figure 8. The percentage of total execution time required by different operations in the Eight-Queens

Question A.2 calls the append vectorized by the compressing method, named c_append.

```
?- c_append(#([a, b], [1, 2, 3], atom),
            #([c], [4, 5], [atom]), Z).                ...... (A.2)
```

The result is Z = #([a, b, c], [1, 2, 3, 4, 5]). Because the execution of the third element has failed, Z contains only two elements (Z is compressed).

Although there are no dead elements in the vectors and there is no need for control vectors, like mask vectors or index vectors, in the compressing method, this method is rather inefficient. This is because,if any vector is compressed, all other vectors which must be used afterwards must also be compressed.

Appendix 2. The Source Program and the IL of the Eight-Queens Problem

The source program and part of the IL is shown here.

A.1 The Source Program

The source program of the Eight-Queens problem is as follows.

```
queen(Q) :- put([1, 2, 3, 4, 5, 6, 7, 8], [], Q).

put([], B, B).
put(Qs, B, Q) :-
    select(Qs, Q1, R), not_take(B, Q1), put(R, [Q1 | B], Q).

select([A | L], A, L).
select([A | L], X, [A | L1]) :- select(L, X, L1).

not_take(R, Q) :-
    Qa is Q + 1, Qs is Q - 1, not_take1(R, Qa, Qs).

not_take1([], Qa, Qs).
not_take1([Q | R], Qa, Qs) :-
    Q =\= Qa, Q =\= Qs,
    Qaa is Qa+ 1, Qss is Qs - 1,   not_take1(Q, Qaa, Qss).
```

A.2 The IL with the Masked Operation Method

Vectorized procedure of not_take1 by masked operation method is as follows.

```
v_not_take1(_, _, _, MI-MI) :-
    v_finished(MI), !.
v_not_take1(B, Qa, Qs, MI-MO) :-
    v_null(B, MI, MO1),
    v_carcdr(Q, R, B, M1, M2),
    'v_=\='(Q, Qa, M2, M3), 'v_=\='(Q, Qs, M3, M4),
    'vs_+'(Qa, 1, Qaa, M4), 'vs_-'(Qs, 1, Qss, M4),
    v_not_take1(R, Qaa, Qss, M4-MO2),
    v_end_or(MO1, MO2, MO).
```

The vectors whose names are prefixed by M are the mask vectors.

Vectorized procedure of select by masked operation method is as follows.

```
v_select(AL, X, Y, MI, BI-BO) :-
    v_select1(AL, X1L, Y1L, MI-ML),
    v_merge(X1L, X, ML), v_merge(Y1L, Y, ML),
    v_repeat(BI, BO, ML).
v_select1(_, [], [], MI-[]).
    v_finished(MI), !.
v_select1(AL, [A' | X1L], [L' | Y1L], MI-[M1' | ML]) :-
    v_carcdr(A', L', AL, MI, M1'),
    v_carcdr(A, L, AL, MI, M1),
    v_select1(L, X1L, L1L, M1-ML1),
    mapcar(v_cons(A), L1L, Y1L, ML1, ML).
```

A.3 The IL with the Indexing Method

Vectorized procedure of not_take1 by indexing method is as follows.

```
x_not_take1(_, _, _, II-#()) :-
    v_finished(II), !.
x_not_take1(B, Qa, Qs, II-IO) :-
    x_null(B, II, IO-IO1),
    x_carcdr(Q, R, B, II, I2),
    'x_=\='(Q, Qa, I2, I3), 'x_=\='(Q, Qs, I3, I4),
    'xs_+'(Qa, 1, Qaa, I4), 'xs_-'(Qs, 1, Qss, I4),
    x_not_take1(R, Qaa, Qss, I4-IO1).
```

The vectors whose names are prefixed by I are the index vectors.

A.4 The IL with the Compressing Method

Vectorized procedure of not_take1 by compressing method

```
c_not_take1(QR, Qa, Qs, B-B, R-R, Q-Q) :-
    v_finished(QR), !.
c_not_take1(QR, Qa, Qs, BI-BO, RI-RO, QI-QO) :-
    v_null(QR, _, M1),
    c_compress(BI, M1, BO-BOe),
    c_compress(RI, M1, RO-ROe),
    c_compress(QI, M1, QO-QOe),
    v_carcdr(Q, R, QR, _, M2),
    'v_=\='(Q, Qa, M2, M3), 'v_=\='(Q, Qs, M3, M4),
    c_compress(R, M4, RI-#()),
    c_compress(Qa, M4, Qa1-#()),
    c_compress(Qs, M4, Qs1-#()),
    c_compress(BI, M4, BI1-#()),
    c_compress(RI, M4, RI1-#()),
    c_compress(QI, M4, QI1-#()),
    'vs_+'(Qa1, 1, Qaa, _), 'vs_-'(Qs1, 1, Qss, _),
    c_not_take(R1, Qaa, Qss, BI1-BOe, RI1-ROe, QI1-QOe).
```

The c_compress procedure makes a vector without dead elements.