A Vectorization Technique for Prolog without Explosion

Yasusi Kanada* and Masahiro Sugaya
Central Research Laboratory, Hitachi Ltd.
Kokubunji, Tokyo, Japan 185

Abstract

This paper describes a technique for executing logic programming languages such as Prolog for the Cray-type vector processors. This technique, which we call the *parallel backtracking technique*, enables a kind of or-parallel execution without process explosion. The compiled intermediate language code for the parallel backtracking execution is the same as tbe code presented in our previous paper. The compilation is based on a kind of program transformation called *or-vectorization*. However, the interpretation of tbe intermediate code is changed to enable the parallel backtracking execution. An execution simulator and a compiler prototype were developed. We bave not yet implemented this technique to our native code execution system, but we expect a performance of eigbt times or more higher than scalar processing upon implementation.

1 Introduction

We [Kan 88a] developed vectorization techniques which enables execution of logic programming languages such as Prolog on pipelined vector processors such as the Hitachi S-810 [Nag 84] or the Cray-2. Using these techniques, a Prolog program is transformed into vectorized program in an intermediate logic programming language; then this program is compiled into procedural programs. The former is called the *vectorization* phase, and the latter tbe *code generation* phase. A type of or-parallelization, which is done in the vectorization phase, enables Prolog programmers to use operation pipelines and storage-access pipelines of vector processors, which leads to the expectation of high performance. We compiled a program of the eight-queens problem by hand using these techniques, and achived a high performance of 4.5 MLIPS on the S-810.

The major drawback of the method described above is that it does not avoid process explosion. In a parallel processing of highly or-parallel program by a naïve method, the number of processes may be increased explosively and the computation may become unable to continue due to resource exhaustion. This situation is called *process explosion*. Though the *eight*-queens problem can be solved using the above method because the number of processes remains within range of computational powers, the *twelve*-queens problem might fail to solve.

This paper describes an execution tecbnique for the vectorized program without explosion. Section 2 overviews the compilation and execution method for vector processors. Section 3 overviews the parallel backtracking technique [Kan 88b], which is a technique for avoiding explosion of vector length in combinatorial searcb. Section 4 describes the technique for avoiding process explosion in vector processing of Prolog. This technique is an extension of tbe parallel backtracking technique. Section 5 draws conclusions about out metbods.

Other works on vectorizing logic programming languages [Nil 86] [Nil 88] [Tat 87] are in the works. There are two major differences between their approaches and ours. One is the difference of the source languages. We use Prolog (in a wide sense), which has and-parallelism and global or-parallelism. They use GHC [Ued 85], which has and-parallelism and only local or-parallelism. The other is the difference in processing structure. Our method is based on compile-time program transformation. Their method is based mainly on interpreters.

2 An overview of Prolog vectorization

There are two kind of concurrent processings. One is *vector* or *pipelined* processing and the other is *parallel* processing. If a computation is fitted well in vector processing, the hardware is very efficiently used. However, vector processing, which is a kind of SIMD-type parallel processing, is more inflexible than MIMD-type parallel processing. Only the same kind of operations can be performed by an instruction. So, not all the programs can be fitted well in vector processing, and a compile-time program transformation is necessary to do so. This transformation is called *vectorization*.

To execute a Prolog program on vector processors, a method of program transformation to fit it in vector processors must be used. *Or-vectorization* [Kan 88a], a type of program transformation, enables a type of or-parallel

*The author's current address is Center for Machine Translation, Carnegie-Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, USA. The e-mail address is yk@a.nl.cs.cmu.edu.

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

Figure 1: A program of the eight-queens problem

execution. In execution of a program such as the eight-queens, a lot of or-processes are generated and the each one's behavior is very similar to others'. So, we can bundle the processes to fit in vector processors.

The same goal (in a static sense) for different processes are bundled in our method. That means that each variable of the source program is replaced by a vector of variables by *or-vectorization*, and each element of the vector is the value of the original variable in each process.

Figure 1 shows an example of the eight-queens program by Nakashima [Nak 83]. However, the four-queens program is used for the example in this paper because the execution of the eight-queens program is complicated. The four-queens program is obtained by replacing the list `[1,2,···,8]` by `[1,2,3,4]` in the first line of the eight-queens program.

Figure 2 outlines the execution of the vectorized four-queens program. `VB` is the vectorized counterpart of variable `B`, which appears in procedure `put` of the source program. Because only one process is generated by the question `?- put([1,2,3,4],[],Q)`, `VB` has only one element initially. This element contains an empty list, which represents an empty chessboard. Four processes are generated by the vectorized counterpart of procedure `select`, so `VB` is reproduced to have four elements, each of which points a single-element list which represents a chessboard with one queen. We omit the explanation of the succeeding part of the execution.

Usually, two or more vectors are processed in a step of a program execution. In the ahove example, a vector of partial solutions, `VB`, and a vector of unused queen list which is the vectorized counter part for `Qs` are processed in the same steps. All the vectors which are processed in the same step have the same number of elements. If the number is N, the value of the i-th element ($1 \leq i \leq N$) of each vector belongs to the same goal (in a dynamic sense) of the source program.

For example, Step 1 of Figure 2 results in the same

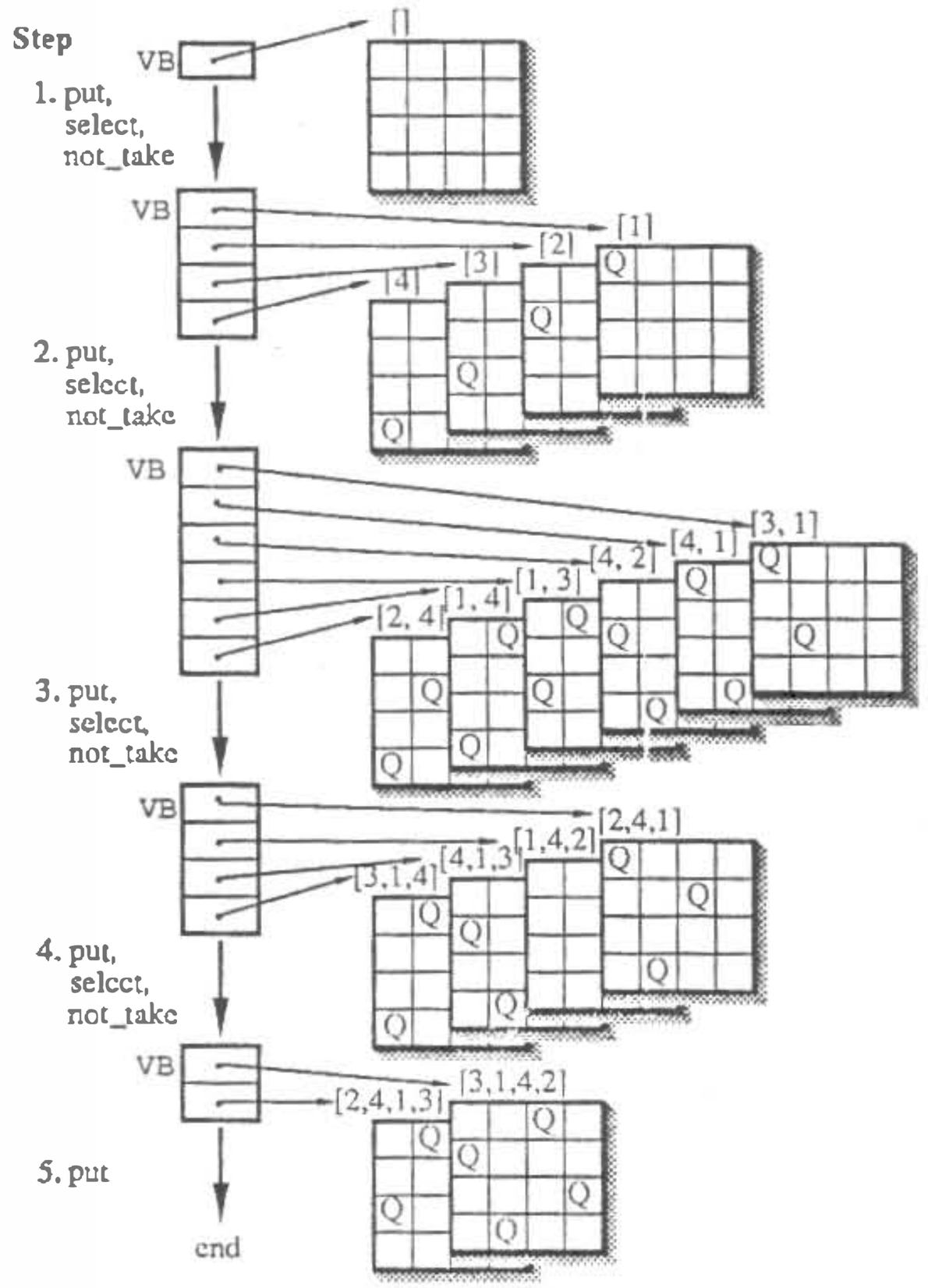

Figure 2: An execution process of the eight-queens program

```
v_select(AL,X,Y,MI,BI,BO) :-
   v_select_1(AL,X1L,Y1L,MI,ML),
   v_merge([X1L,Y1L],[X,Y],[BI],[BO],ML).

v_select_1(_,[],[],MI,[]) :-
   v_finished(MI), !.
v_select_1(AL,[A'|X1L],[L'|Y1L],
           MI,[M1'|ML]) :-
   v_carcdr(A',L',AL,MI,MI'),
   v_carcdr(A,L,AL,MI,M1),
   v_select_1(L,X1L,L1L,M1,ML1),
   map_v_cons(A,L1L,Y1L,ML1,ML).

map_v_cons(_,[],[],[],[]).
map_v_cons(A,[XH|XL],[YH|YL],
           [MIH|MIL],[MOH|MOL]) :-
   v_cons(A,XH,YH,MIH,MOH),
   map_v_cons(A,XL,YL,MIL,MOL).
```

Figure 3: The definition of `v_select` in the intermediate logic programming language

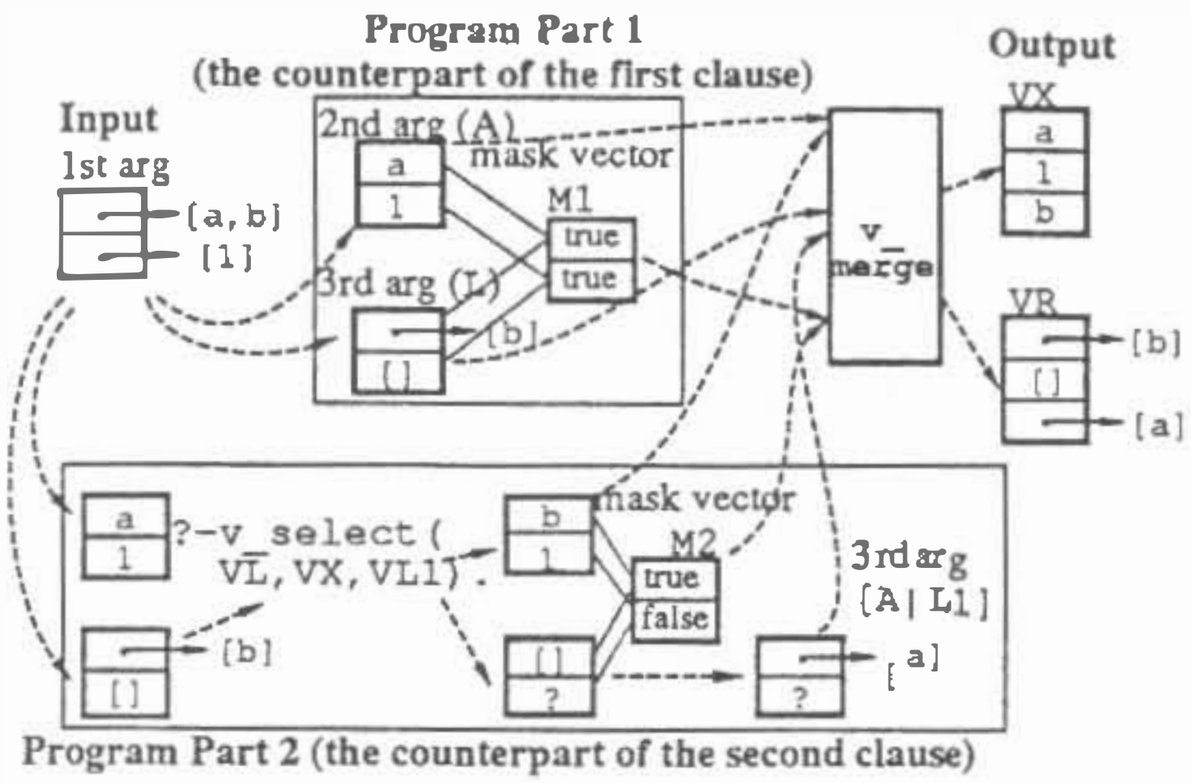

Figure 4: Vectorized execution process of procedure `select`

solutions as executing the following four goals in parallel:

```
?- put([2,3,4],[1],Q1).
?- put([1,3,4],[2],Q2).
?- put([1,2,4],[3],Q3).
?- put([1,2,3],[4],Q4).
```

The vectorized form of these goals is as follows:

```
?- v_put(#([2,3,4],[1,3,4],
           [1,2,4],[1,2,3]),
         #([1],[2],[3],[4]),
         #(Q1,Q2,Q3,Q4)).
```

$\#(e_1, \cdots, e_n)$ represents a vector whose elements are $e_1, \cdots, e_n$. Tbe length of all the vectors are four here. The vector shown in Figure 2 is the first argument of the above vectorized goal.

The details of the transformation techniques and examples are described in the previous paper [Kan 88a].

During execution of deterministic procedures, procedures which have one or no solution, the vector length, or the number of elements in each vector, is constant. Some goals may fail, and the number of the valid elements decreases. However, vectors with *dead* elements can be processed with masked operation facility or list vector facility of vector processors [Kan 88a]. So the number of the elements can be constant in the *or-vectorized* program.

On the contrary, the vector length varies during the execution of nondeterministic procedures. More than one solutions are generated from each vector element, and the solutions (the values of each variable) should be accumulated into a vector to lengthen the vector length. The reason why they should be done so is as follows. Vector processors are slower than scalar processors when they are used with single-element vectors only. High performance is achieved when using them wi th vectors of sufficient length, namely one hundred or more elements. In most programs with or-parallelism, there is only one process initially. So, if the solutions are not accumulated, the vector length will be always one, and the performance cannot be improved.

The solutions are accumulated by the following mechanism. Each vectorized program part, which is a counterpart of one of the clauses of the nondeterministic procedure, is executed separately. Unlike normal or-parallel execution, these parts are usually executed sequentially. Each part generates a vector for each variable of the source program. The control is not transfered to the outside of the procedure until the execution of all the clauses are finished. At the end of the procedure execution, all the vectors for the same variable of the source program are merged into a single vector.

For example, the execution of the following program, wbich is a part ofthe eight-queens program, is explained:

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

The intermediate code of `select` is very similar to thatof nondeterministic procedure `append`, which is described in the previous paper [Kan 88a]. So only a brief explanation about the vectorized code of `select` is presented here. Figure 3 shows the intermediate code of `v_select`, and Figure 4 outlines the execution of the following vectorized goal:

```
?- v_select(#([a,b],[1]),VX,VR).
```

In the source program, `select` inputs a list and selects one of the elements and returns it and the list of the rest elements. So, question `?- select([a,b],X1,R1)` returns the following two solutions:

```
X1 = a, R1 = [b].
X1 = b, R1 = [a].
```

Question `?- select([1],X2,R2)` returns the following one solution:

```
X2 = 1, R2 = [].
```

Procedure `v_select` is the vectorized counterpart of procedure `select`. The first argument of `v_select` is the input. In the above vectorized goal, the first argument

is a vector of two elements, [a,b] and [1]. So this goal is the vectorized counterpart of the above two questions.

The body of procedure `v_select` consists of three parts. The first part corresponds to the first clause of procedure `select`, and the second part corresponds to the second clause. The third part has no counterpart in the source program.

The first part computes the two solutions, and returns the vectors with these solutions. The second part computes the rest solution, calling `v_select` recursively, and returns the vectors of the solutions. The *live-ness* of the vectors is displayed by a *mask vector* [Kan 88a], `M1`. Both elements of `M1` are `true`, that means all the elements of the vectors are *live*, solutions. The second part returns vectors of two elements, but the second elements are *killed* or invalidated. The *live-ness* of the vectors is displayed by mask vector `M2`.

The third part, procedure `v_merge`, inputs the outputs of the previous parts, merges them, and outputs two vectors which contain all of the solutions. The result of the whole computation is as follows:

```
VX = #(a,  1, b).
VB = #([b],[],[a]).
```

The length of all the vectors before `v_merge` are the same, two. The vector length is changed only by `v_merge` and the lengths of all the vectors outputted by `v_merge` are the same, three.

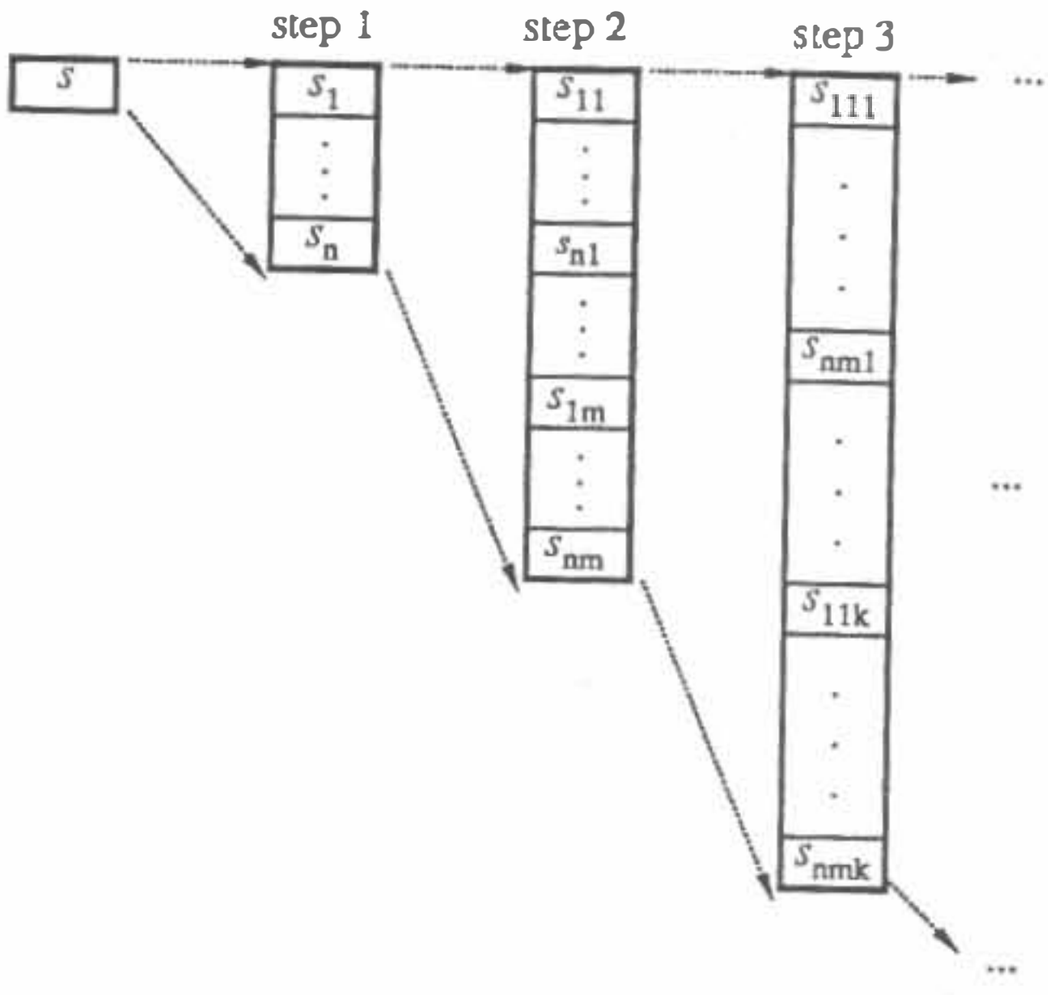

Figure 5: Vector length explosion in a combinatorial search by vector processors

3 The parallel backtracking technique

The parallel backtracking technique avoids explosion of vector length in parallel processing of combinatorial search programs [Kan 88b]. It was a technique for vectorizing procedural backtracking programs, though it was expected to be applied for vectorizing Prolog programs [Kan 85].

Figure 5 explains a problem of simple vector processing of search problems. In a vector processing of search problems such as the eight-queens by vector processors, the vector length increases because the number of possible solutions, each of which is an element of the vector, explosively increases during processing. The vector elements may overflow from the main storage or the disks, then the computation fails to continue.

Figure 6 explains the solution for the problem given in Kanada [Kan 88b]. The vector is split into two or more small vectors when the vector length becomes too large (step 2 and 2'). Then the computation is continued only for one of the vectors (V_1) and the others are saved as *choice points*. When the computation for the first vector finishes (step k), the control returns to one of the choice points and the same computation as for the first vector (V_1) is performed for the second vector (V_2). All the split vectors are processed in the same way. In the case of Figure 6, V_2 is the final vector. The vector splitting may be done more than once, as shown in Figure 6 (step k+2 and k+2').

Whole computation is done in *or-parallel* in the process shown in Figure 5. On the other hand, some "backtrackings" occur in the process shown in Figure 6, which

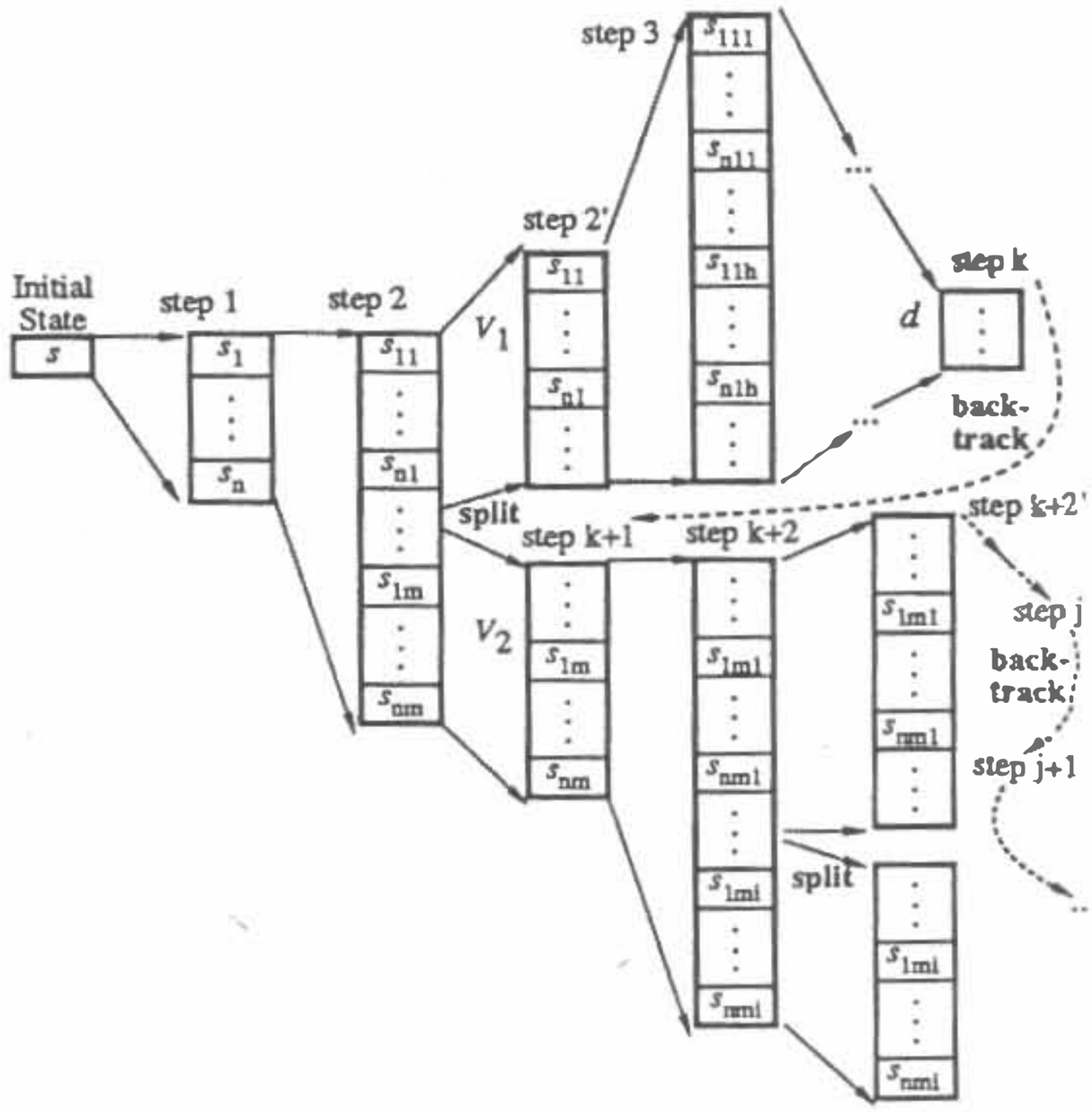

Figure 6: A combinatorial search using parallel backtracking technique

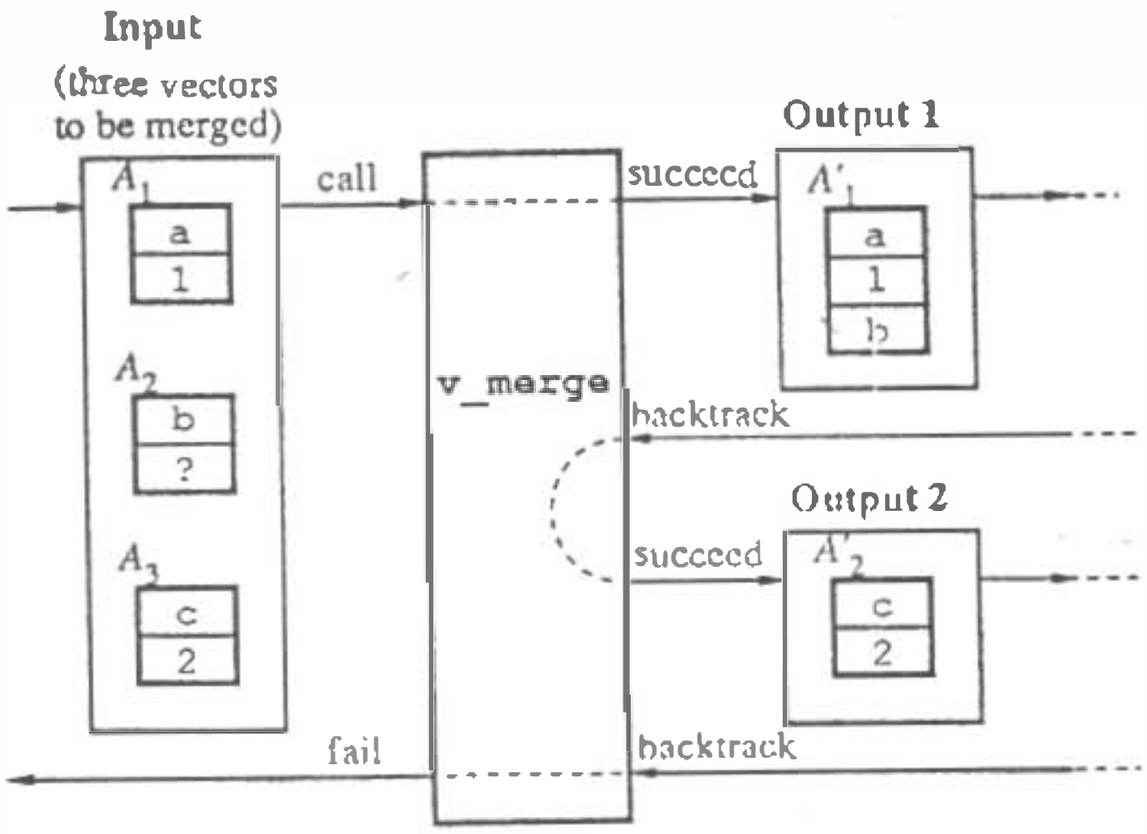

Figure 7: An execution example of parallel backtracking `v_merge`

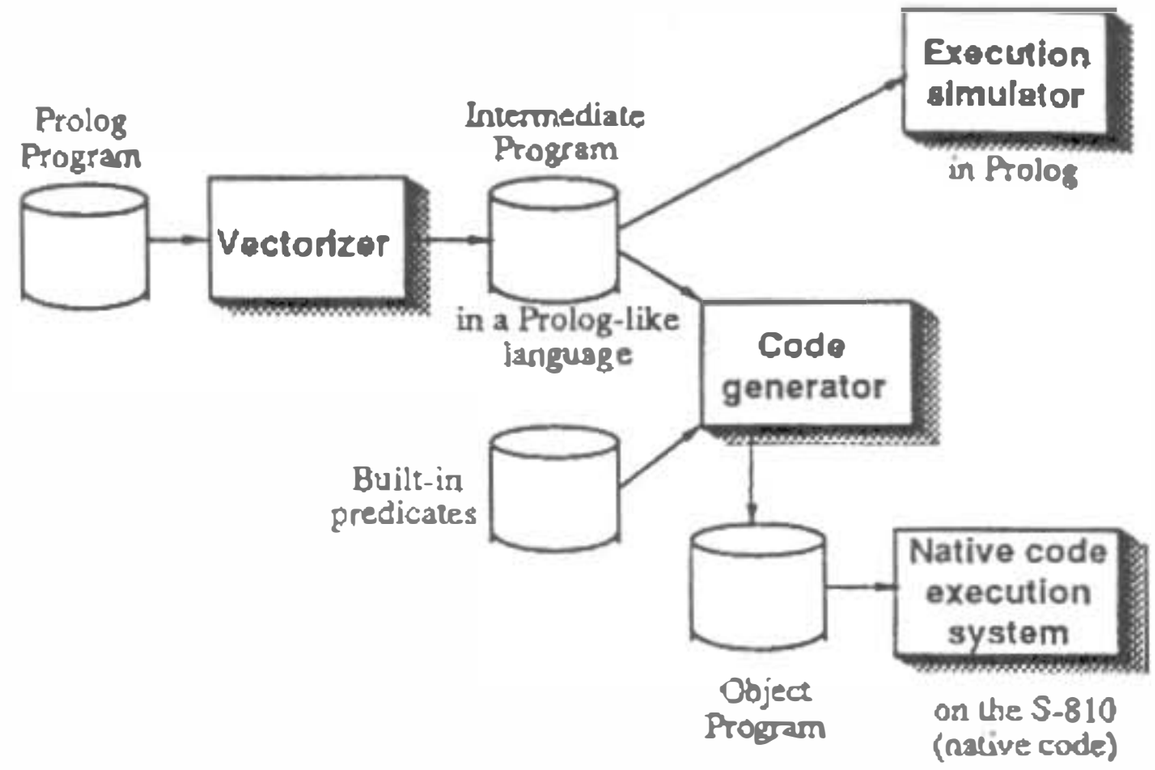

Figure 8: The process of Prolog compilation and execution for vector processors

is the reason why this technique is called the "parallel backtracking technique."

4 Explosion-free execution of Prolog

The explosive increase of vector elements is very close to the process explosion in or-parallel processing of Prolog. If the parallel backtracking technique described in the last section is slightly modified, it is applicable to the vectorized execution of Prolog, thereby avoiding a process explosion without the expense of efficiency. This section explains the method of applying the parallel backtracking technique to Prolog.

First, the program points, where the vectors should be split and choice points are made, must be decided to apply this technique. Vectors are split only in `v_merge`, and choice points are made only in `v_merge` in our parallel backtracking execution. Because the vector length is increased in the execution of procedure `v_merge` of the intermediate language, and it is kept constant in other execution steps in the execution method shown in our previous paper [Kan 88a]. Procedure `v_merge` outputs the input vectors as they stand if the vector lengths are sufficient, or merges them if not sufficient. This merge may be a partial merge, i.e. merging some of the vectors but not all.

Figure 7 shows an execution example of `v_merge` under its parallel backtracking interpretation. Procedure `v_merge` inputs three vectors to merge, A_1, A_2 and A_3, outputs two vectors, A'_1 and A'_2, and an input mask vector which is not shown in Figure 7. The input vectors of `v_merge` are formed into list of vectors, so `v_merge` actually inputs one list of vectors and outputs one vector. The first and the second input vectors are merged and the resulting vector is outputted, and the third one is outputted as it stands. This means `v_merge` does a partial merge here. A *dead* element in A_2, which is displayed by the input vector, is not included in A'_1.

When `v_merge` is called, it succeeds and outputs the first result, A'_1, and it makes a choice point for A'_2. When a backtracking to the choice point in `v_merge` occurs, `v_merge` outputs the second result, A'_2. Since the intermediate language is Prolog-like, backtrackings are handled appropreately. A more backtracking to `v_merge` just causes a failure because there is no more choice point, and it causes a further backtracking.

This functional change of `v_merge` can be done without syntactic changes of the intermediate code. Only the interpretation of the intermediate code is changed. No global choice points are created in the interpretation given in our previous paper, but choice points are created and global backtrackings to them are occurred in the new interpretation. So the intermediate language may have been a procedural one in the previous paper, but it must be a Prolog-like language with automatic backtracking for applying the parallel backtracking technique.

There are two methods for merging and splitting vectors. One is *to merge first then to split*, and the other is *to merge only when vector length is short*. The former method merges all the input vectors first and then splits them appropriately. The good point of this method is that vector length can be chosen arbitrary. However, this method is inefficient because both merging and splitting require the copying of vectors. The latter method merges vectors only when they are too short and it does not actually split vectors. The latter method is probably better because it require less copying of vectors. In the latter method, it is probably good to merge the input vectors one by one until the vector length becomes sufficient for achieving high performance by vector processors.

We have developed a simulator for vectorized execution, and a compiler which inputs a Prolog program with mode declarations and generates vectorized intermediate language program. Both programs are written in Prolog. The structure of this system is outlined in Figure 8. This intermediate language (IL) is a Prolog-like language with vectors. Its syntax is the same as the IL in our previous paper. The simulator inputs the IL, and computes the solutions using vectors, which are simu-

lated by functor #(···), a functor whose name is "#", in Prolog. A functor is good for simulating vectors because elements of a functor can be accessed by indices using built-in predicate arg in Prolog, and vector elements are immutable (unassignable) in our model. Each build-in procedure in the intemediate language, such as v_merge, is simulated by a Prolog procedure. So the intermediate code is executed directly by the simulator.

Though the computation is performed in pseudo-parallel, the simulator decomposes resulting vectors and returns the solutions one by one to the user. So, the user interface is very similar to Prolog's, though the order of solutions may be different from sequential Prolog's. For example, we assume that the user types a question and the first vector of solusions of the program, S_1, contains two solutions, s_{11} and s_{12}. It will take a little time until s_{11} is printed, because it requires to compute the value of S_1 and to extract s_{11} from S_1. However, if the user types ";", or requires a backtracking, s_{12} will be printed immediately because it only requires to extract it from S_1. If the user requires a more solution, the user may have to wait a little because it requires to compute the second vector of solutions, S_2.

Though Prolog is used, most part of the simulator program is deterministic; no choice points are made in most part. However, there are two procedures where global backtracking is used. One is procedure v_merge, and the other is the procedure which decomposes vectors for the user.

5 Conclusion

This paper has shown a technique for executing Prolog programs on vector processors without process explosion. This technique is called the parallel backtracking technique. We developed a execution simulator using this technique, which inputs intermediate code generated by a vectorizing compiler described in our previous paper [Kan 88a].

We have not yet developed a real execution system. However, using this technique, we can certainly make a Prolog system whose performance ranks more than eight times higher than the mainframe machine in solving the eight-queens program. Because the non-backtracking execution of this program achieves a high performance of nine times higher than scalar processing, and backtrackings does not take much time in the parallel backtracking execution.

Acknowledgement

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

References

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

[Kan 88a] Kanada, Y., Kojima, K., and Sugaya, M.: Vectorization Techniques for Prolog, *ACM International Conference on Supercomputing*, 1988.

[Kan 88b] Kanada, Y., and Sugaya, M.: A Schema of Solving Search Problems on Vector Processors: Parallel Backtracking Schema, *Transaction of Information Processing*, Vol. 29, No. 10, 1988 (in Japanese).

[Nag 84] 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.

[Nak 83] Nakashima, H.: Introduction to Prolog, Sanpoh-Shuppan, 1983 (in Japanese).

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

[Nil 88] Nilsson, M., and Tanaka, H.: A Flat GHC Implementation for Supercomputers, *Fifth International Symposium on Logic Programming*, 1988.

[Tat 87] Tatsuguchi, Y., and Muraoka, Y.: An Implementation of a Parallel Logic Programming Language on a Vector Processor, *35th National Conference of Japan Society of Information Processing*, 5Q-1, pp. 753-754, 1987 (in Japanese).

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