논리 프로그램의 병렬도 개선을 위한 플랫 인덱싱 기법

김 희 철† · 이 용 두†

요 약

본 논문은 논리언어 프로그램의 효율적인 클로즈(Clause) 인덱싱을 위한 컴파일 기법에 대한 체계적인 접근방법을 제시한다. 본 접근방법의 핵심으로서 노드당 평균 병렬도와 클로즈 수행시도(clause trial) 횟수를 정확하게 나타낼수 있는 기법으로서 인덱싱트리(Indexing Tree)를 제안한다. 인덱싱트리는 인덱싱 수행 시에 인덱싱을 위한 지시어(Instruction)의 수행 결과로 프로그램의 컨트롤이 실패처리코드로 이동하는 경우도 정량적으로 나타내 준다. 인덱싱트리를 사용하여 논리 프로그램을 위한 대표적인 가상머신인 WAM(Warren Abstract Machine)을 분석한 결과 WAM에서 사용하는 인덱싱 기법이 논리 프로그램의 병렬 처리에 있어 탐색트리의 병렬도를 감소시키며, 또한 스케쥴링의 효율성을 저하시키는 결점을 내포하고 있음을 발견할 수 있었다. 이러한 결점을 해결하기 위하여 본 논문은 플랫 인덱싱이라는 새로운 인덱싱 기법을 제안하고 이것을 실제 논리언어 컴파일러에 구현하여 측정한 향상 및 분석 결과를 보여준다.

Flat Indexing: A Compilation Technique to Enhance the Parallelism of Logic Programs

Hiecheol Kim† · Yong-Doo Lee†

ABSTRACT

This paper presents a systematic approach to the compilation of logic programs for efficient clause indexing. As the kernel of the approach, we propose the indexing tree which provides a simple, but precise representation of average parallelism per node (i.e., choice point) as well as the amount of clause trials. It also provides the way to evaluate the number of the cases that the control is passed to the failure code by the indexing instruction such as *switch_on_term*, *switch_on_constant*, or *switch_on_structure*. By analyzing the indexing tree created when using the indexing scheme implemented in the WAM, we show the drawback of the WAM indexing scheme in terms of parallelism exposition and scheduling efficiency. Subsequently we propose a new indexing scheme, which we call the Flat indexing. The experiment result shows that over one half of the benchmarks benefit from the Flat indexing such that, compared with the WAM indexing scheme, the number of choice points is reduced by 15%. Moreover, the amount of failures which occurs during the execution of indexing instructions is reduced by 35%.

1. Introduction

Logic languages[13,14], based on the SLD refutation[15], impose a strictly sequential search over its search tree. For a given goal, clauses making up its predicate are *tried* sequentially in their textual order. In each *clause trial*, unification occurs between the clause's and the goal's arguments. If the

※ 이 논문은 1998학년도 대구대학교 학술연구비 지원에 의한 논문임.

† 정 회 원 : 대구대학교 정보통신학부 교수

논문접수 : 1998년 2월 23일, 심사완료 : 1998년 4월 13일

arguments of the clause are variables, the clause must be tried regardless of the type of the goal arguments because the clause is always successfully unified. On the other hand, provided some arguments are non-variables, their information allows us to determine the clauses which will always fail to unify; those clauses can sometimes be pruned away from the list of clauses to try. The technique is usually called *indexing* and is clearly a good way to improve the performance when the number of clauses making up a predicate is large. Therefore, compilers for most logic language produce the code which implements a kind of such indexing[1,8].

In spite of the research efforts over the last decade, the efficient implementation of Prolog is still an important issue in the logic programming. As a sequential engine for logic languages, the WAM(Warren Abstract Machine) is one of breakthroughs which contribute to a highly efficient implementation of logic programs[16]. Although its memory organization and instructions are slightly modified or extended to support parallelism, the WAM is used in most parallel implementations of logic languages as their sequential engine to achieve high single thread performance[4,9].

A choice point is one of runtime data structures created during the execution of the WAM code [1,16]. It serves to keep the information associated with the execution of a goal (*i.e.*, a predicate). Although according to the role of the choice point, it is natural to provide a single choice point per each invocation of a predicate, the WAM compilers create two choice points appearing contagiously in a search path. But, these two choice points do not cause any significant disadvantage against the sequential WAM except for the cost paid for the creation of an additional choice point. Rather, they contribute to very compact code, which is in fact one of the design objectives of the WAM[16].

In the parallel execution of the WAM code, choice points have an additional role to represent OR-parallelism[9]. This causes the issue of the two choice points to become very controversial in terms of parallelism and scheduling. Parallel schedulers exploit OR-parallelism by taking available (OR-parallel) branches (*i.e.*, *clauses*) from choice points[3],[9]. Because OR-parallelism of a predicate can be crudely regarded as the number of alternative clauses which make up the predicate, the available OR-parallelism spreads over the two adjacent choice points, provided two choice points are created. In this case, the available OR-parallelism is not fully exposed at the point of the goal invocation; only a part of OR-parallelism is available at the point of the goal invocation. As a result, the creation of two choice points introduces a harmful effect on the parallelism exposition by decreasing the average OR-parallelism per node.

This paper points out that the creation of those two choice points stems from the indexing scheme implemented in the WAM (which we will call the WAM *indexing*). Through the quantitative analysis, we evaluate the amount of the overhead from the viewpoint of average OR-parallelism per node. To avoid the overhead, we suggest a new indexing scheme which we will call the *Flat indexing*. To verify its performance, we implement both the WAM and the Flat indexing and evaluate respectively the number of choice points created for a set of benchmarks.

The rest of the paper is organized as follows. Section 2 gives a brief introduction of the WAM indexing scheme. Section 3 presents a framework used for the analysis of indexing schemes as well as the result of the WAM indexing obtained by applying the framework. Section 4 presents the Flat indexing that we

propose to enhance the OR-parallelism and scheduling efficiency. Section 5 reports the evaluation result. Finally, section 6 offers conclusions and future researches.

2. Background

This section introduces the WAM indexing to make the paper self-contained as well as to introduce some terminology to be used in subsequent sections. In the WAM indexing, the *first argument* of either a clause or a goal is used as the key[1], [16]. According to the usual programmer's tendency, clauses making up a predicate are usually defined differently depending on data types. The differentiation is mostly reflected in the first argument. Considering the trade-off between efficiency and simplicity, the usage of the *first argument* as the *key* for indexing can be considered as a quite reasonable choice.

Given an input key (*i.e.*, the *first* argument of a goal), the WAM indexing is applied to the predicate to prune out a subset of the clauses before their clause trials, provided those clauses shall always fail the unification. Among the four data types, (*variable*, *constant*, *list*, and *structure*), if a clause has a variable as its *clause key* (*i.e.*, the *first* argument of the clause), the clause should not be pruned away because the clause key will always unify with the *input key* of any type.

```
c1: match(sum(A,B), sum(C,D)) :-
    match(sum(A+D-1), sum(C+B-1)).
c2: match(sum(A,B),C) :-match(B-1,sum(c,B-1)).
c3: match(a,b) :-match(numeric(a), numeric(b)).
c4: match(X,ascii(Y)) :-match(ascii(X),digit(Y)).
c5: match(a,b) :-match(ascii(a),ascii(b)).
c6: match(a,b) :-match(digit(a),digit(b)).
c7: match(b,X) :-match(digit(b),digit(X)).
c8: match(sum(A,B), sum(C,D)) :
    -equal((A-C), equal(D-B)).
c9: match(sum(A,B),C) :-match(sub(C-A),B).
c10: match([A,B],[C,D]) :-match(A,C), match(B,D).
c11: match([a,A],[C,b]) :-match(a,C), match(A,b).
c12: match(X, numeric(Y)) :
    -match(numeric(X),numeric(Y)).
```

(Fig. 1) Clauses making predicate match/2

```
match_2:    try_me_else G2_label
            [Code for G1]
G2_label:   retry_me_else G3_label
            [Code for G2]
G3_label:   retry_me_else G4_label
            [Code for G3]
G4_label:   trust_me_else_fail
            [Code for G4]
```

(Fig. 2) The structure of the WAM code implementing the chaining of four groups in match/2

To make such clauses always subject to clause trials, the WAM indexing divides the clauses, $(c_1,\ldots,c_m)$, which make up a predicate, into a set of groups, $G_1,\ldots,G_m$ $(1 \le m \le n)$, where G_i is a set of contiguous clauses either of the following two types:

- Type α: G_i consists of only a single clause whose key is a variable
- Type β: G_i consists of a maximal sequence of contiguous clauses whose key is not a variable.

Fig. 1 shows twelve clauses which define predicate *match/2*. According to the above grouping rule, four groups are defined as follows: $G_1 = \{c_1, c_2, c_3\}$, $G_2 = \{c_4\}$, and $G_3 = \{c_5\ c_6, c_7\ c_8, c_9, c_{10}, c_{11}\}$, $G_4 = \{c_{12}\}$. Note that G_2 and G_4 are of type α and G_1 and G_3 are of type β.

In the WAM code of a predicate, groups defined for the predicate are chained such that

each group is visited consecutively at runtime regardless of the type of the input key. By doing this way, the clauses with variable keys are always tried. The implementation uses instructions *try_me_else*, *retry_me_else*, and *trust_me_else_fail*[16] as shown in Fig 2.

It is not necessary to make any indexing as for groups of type α, because they contain only a single clause which must be always tried. The actual indexing is applied to the groups of type β that consist of more than one clauses. The indexing consists of two steps made firstly with respect to *data types* of the clause key, and secondly with respect to *their values*. The latter step is applied only to the clauses whose keys are of either the constant or the structure type because only constant or structure data can have multiple different values.

Given a group of type β, the indexing process begins with making partitions with respect to the clauses of the group according to the data type of the clause key. As the key of each clause has one of the following three data types, (*constant*, *list*, and *structure*), three *partitions* are produced respectively as P_c, P_l, and P_s, where P_c, P_l, or P_s, is an ordered set of clauses whose keys are respectively a constant, a list, or a structure. In addition to these partitions, P_v is defined as the ordered set of all the clauses in the group. As the second step of the indexing, one more level of partitioning is made as to P_c, and P_s. In this step, the clauses having the same key value are grouped as a subpartition.

Given an input key of type x, the indexing made over the four partitions, (P_c, P_l, P_s, P_v), selects one of the partitions, P_x, according to the type of the input key. The control is then dispatched to partition P_x. By doing this, the rest of the partitions are excluded from clause trial. The implementation is made by using instruction *switch_on_term* that has four arguments provided respectively for each type. The value of each argument has one of the following values.

- Case 1: the address of the clause's code (*e.g.*, C_i in Fig 3) when the relevant partition contains only one clause.
- Case 2: the address of the partition's code (*e.g.*, C_partition in Fig 3) when the relevant partition contains more than one clauses.
- Case 3: the address of the code(*e.g.*, fail in Fig 3) which processes the unification failure when no partition is defined for the given type.

When the partition thus selected from instruction *switch_on_term* is either P_c or P_s, the control is dispatched to some subpartition according to the value of the input key. The implementation is made by using instruction *switch_on_constant* (or *switch_on_structure*). Either instruction has a hash table as its argument in which each subpartition is provided with an entry.

```
Gi_code:      seitch_on_term Ci1, Cia|C_partition|fail,
              Cib|L_partition|fail, Cic|S_partition|fail
C_partition:  switch_on_constant {pointers to buckets}
              [Lists of subpartitions for constants]
L_partition:  [A bucket for lists]
S_partition:  switch_on_structure {pointers to
              subpartition}
              [Lists of subpartitions for structures]
Ci1_label:    try_me_else Ci2_label
              [Code for clause Ci1]
Ci2_label:    retry_me_else Ci3_label
              [Code for clause Ci2]
...           ...
Cik_label:    retry_me_else_fail
              [Code for clause Cik]
```

(Fig. 3) The code structure of a group which has k clauses. Argument C_{ia}/C_partition|fail *indicates either* C_{ia}, C_partition *or fail.*

The entry holds the address of the subpartition's code if the subpartition contains more than one clauses; otherwise, it holds either the address of the clause if there is a clause or the address of the failure code if there is no clause matching with the input key. Therefore, for a given input key, according to the value of the entry, the control is dispatched either directly to a clause, to a subpartition, or to the *failure code*.

The partition, selected when the input key is a list or a variable, as well as the subpartition selected when the input key is a constant or a structure will be referred to as a *bucket* to be denoted as B_x where x is either *l*, *v*, *c*, or *s* to indicate the type of the input key. According to the earlier description of switching instructions (*switch_on_term*, *switch_on_constant*, or *switch_on_structure*), it should be noted that a bucket always contains more than one clauses. The WAM code of a bucket is organized such that all the clauses in the bucket are tried one by one. The implementation is made by using instructions *try*, *retry*, and *trust*[16]. Fig 3 shows the code produced for group G_3 of *match*/2.

```
C_partition          switch_on_constant 2,
                     (a: Ca_subpartition, b: C7_Code)
Ca_subpartition:     try C5_Code
                     trust C6_Code
L_partition:         try C10_Code
                     trust C10_Code
                     trust C11_Code
S_partition          switch_on_structure 1,
                     (sum/2 : Ssum_partition)
Ssum_subpartition:   try C8_Code
                     trust C9_Code
C5_Label:            try_me_else C6_Label
C5_Code:             [Code for match(a,b) :-
                     match(ascii(a), ascii(b)).]
C6_Label:            retry_me_else C7_Label
C6_Code:             [Code for match(a,b) :-
                     match(digit(a), digit(b)).]
                     .
                     .
                     .
C11_Label:           trust_me_else_fail
C11_Code:            [Code for match([a,A],[C,b]]) :-
                     match(a,C), match(A,b)]
```

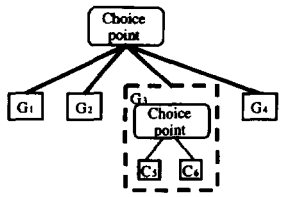
(Fig. 4) A code for group G3 of match/2

Before we proceed to the next section, let us briefly describe the relation between the WAM indexing and the number of choice points created for each invocation of a predicate. In the WAM, instruction *try_me_else* or *try* creates a choice point. When more than one clauses are defined for a predicate, the code for the first clause always starts with instruction *try_me_else*. In this case, if there is a bucket with more than one clauses in any of these groups, its code starts with instruction *try*. If the bucket is sel-ected at runtime, two choice points are thus created contagiously in the search path respect-ively by *try_me_else* and by *try*. For example, instruction *try_me_else* in the code for predicate *match*/2 creates a choice point for which four partitions are exposed as alternative branches (Fig 5). In the sequential execution, the four partitions are executed sequentially from left to right. In parallel execution, they can be exe-cuted in parallel respectively by different proc-essors. As depicted in the figure, if the input key is *a*, a new choice point is created inside group G_3 by instruction *try* in C_a*_subpartition* (Fig. 4).

3. Framework for Indexing Schemes

In the sequential execution of the WAM code, the creation of two contiguous choice points for a predicate (*i.e.* for the execution of a goal) can be regarded just as a variance of an implementation since it does not cause any

significant performance penalty. In the parallel execution, however, it has a very harmful influence on the performance by decreasing the amount of parallelism per choice point and the efficiency of task scheduling. This section presents an analysis framework which aims at identifying the influence that the WAM indexing has on the OR-parallelism of Prolog programs. The analysis consists of the identification of the shape of an OR-parallel search tree created under the WAM indexing and also a quantitative evaluation of the amount of OR-parallelism per choice point.



(Fig. 5) Choice points and parallelism

3.1 Analysis Framework: Indexing Tree

For a predicate P defined by more than one clauses, we now provide some notations and definitions associated with the WAM indexing. Let the set of clauses which make up predicate P be $c_1, \ldots, c_n$. Suppose that the clauses are divided into m groups: $G_1, G_2, \ldots, G_m$. Let us define a mapping N such that $N(S)$ be the number of elements of a set S. The number of clauses in group G_i is then denoted as $N(G_i)$, (*i.e.*, $\sum_{i=0} N(G_i) = n$). For a group G_i, let the buckets selected for each type of the key be b_{iv}, b_{il}, b_{ic}, and b_{is}. If no bucket is selected for a given type x, b_{ix} is regarded as an empty set. Note that a bucket is for a specific value of the key and thus normally includes a subset among the clauses whose key is of its type. Therefore the following relation holds.

$$N(G_i) \le N(b_{il}) + N(b_{ci}) + N(b_{si})$$

In order to analyze how many choice points are created in the execution of a goal, a tree to be called an *indexing tree* is proposed. An indexing tree is a tree $T=(N,E)$, where N is the set of nodes and E is the set of edges. A non-leaf node in an indexing tree represents a choice point and is referred to as a *CpNode* (Choice Point Node). Given a predicate, if the number of its groups m is bigger than one, a choice point is always created at the beginning of its execution. The *CpNode* of this case becomes the root of the predicate's indexing tree and it has m subtrees, each respectively for a group. To see the shape of a subtree, it must be noted that for each group, one of the following three cases occur when executing *switch_on_term*, *switch_on_constant*, or *switch_on_structure*:

- Case 1: No one bucket is chosen; no clause is tried and the control moves directly to the failure code. The subtree of this case is represented as a leaf node to be classified as a *SfNode* (*Switch failure Node*).
- Case 2: A bucket having only one clause is chosen; only one clause is tried. The subtree of this case is represented as a leaf node to be classified as a *CtNode* (*Clause Trial Node*).
- Case 3: A bucket β having more than one clauses is chosen; more than one clauses are tried. The subtree of this case consists of more than one nodes. The root of the subtree is always a *CpNode* because the number of clauses in bucket β is bigger than one. As $N(b)$ clauses will be tried with respect to this new *CpNode* and their trials will not cause any more creation of choice points, there are $N(b)$ *CtNodes* in the subtree.

According to the above discussion, any

indexing tree defined for a predicate has up to three levels and a non-leaf node is always a *CpNode* and a leaf node(*LfNode*) is always either a *SfNode* or a *CtNode*. Fig 5 illustrates an indexing tree for a predicate.

Given a predicate, its indexing tree is defined differently depending on the input key. If input key is a variable or a list, the form of the index tree is always the same regardless of the values of the input key. On the other hand, if the input key is a constant or a structure, the form becomes different depending on the value of the input key. In this case, to compare the size of indexing trees, we define the following criteria:

- Given indexing trees T_1 and T_2, T_1 is larger than T_2 if the number of *CpNodes* in T_1 is bigger than the one in T_2.
- If T_1 and T_2 have the same number of *CpNode*, the one which has more leaf nodes is *larger* than the other.

Based on the above definition, we introduce the maximum and the minimum indexing tree with respect to a given data type as follows.

- For a given data type, the maximum indexing tree of the type is the one which has the largest among the indexing trees possible for the type.
- For a given data type, the minimum indexing tree of the type is the one which has the smallest among the indexing trees possible for the type. When the input argument is either a constant or a structure, the minimum indexing tree is the one resulting from the input key which does not match with any of the clause keys.

3.2 The Analysis Result of the WAM Indexing

In order to analyze the WAM indexing, we use the framework established in the previous section. In the analysis, we calculate for each data type the size of the minimum and the maximum indexing tree of a predicate, where the size is represented in terms of the number of *CpNodes* and *LfNodes*.

Consider a predicate consisting of m groups. The minimum and the maximum indexing tree are the same if the input argument is either a variable or a list; otherwise, they may be different from each other. For a given input key of the constant or the structure type, if the bucket chosen inside a group G_i consists of more than one clauses, let us denote the bucket as b_i. For all $i(1 \le i \le m)$, let r be the number of such buckets. Given an input key, number r is always defined uniquely. When the input key is of the constant (*resp.* structure) type and has the value which results in the maximum value for r, the indexing tree of each case becomes the maximum indexing tree for the constant (*resp.* structure) type. In this case, let us denote each of the bucket in group G_i $(1 \le i \le m)$ as b'_{ic} (*resp.* b'_{is}).

Table 1 shows the analysis result made for a predicate, where p stands for the number of choice points created with respect to the groups. In other words, it becomes *0* if the number of the groups is *1*; otherwise, it becomes *1*. As discussed earlier, r is the number of buckets selected in all the groups with respect to a given input key. Now that a choice point is created for each of the buckets, the total number of *CpNodes* becomes p plus r.

The number of the *LfNodes* of an indexing tree can be obtained by summing up the *LfNodes* of each level. The number of groups minus the number of *CpNodes* (*i.e.*, $m-r$) becomes the number of *LfNodes* in the second level. The number of *LfNodes* in the third level is the total number of clauses in the buckets selected in each partition for the input key.

〈Table 1〉 The minimum and the maximum indexing tree (p is 0 if m=1; otherwise, p is 1)

Type of input key	*Minimum Indexing Tree*		*Maximum Indexing Tree*	
	Number of CpNodes	*Number of LfNodes*	*Number of CpNodes*	*Number of LfNodes*
Variable	p+r	n	p+r	n
List	p+r	$m-r+\sum_{i=1}^{r} N(b_{il})$	p+r	$m-r+\sum_{i=1}^{r} N(b_{il})$
Constant	p	m	p+r	$m-r+\sum_{i=1}^{r} N(b'_{ic})$
Structure	p	m	p+r	$m-r+\sum_{i=1}^{r} N(b_{is})$

〈Table 2〉 The number of clauses (NoB: number of buckets)

Data type	G_1(Type β)		G_2(Type α)		G_3(Type β)		G_4(Type α)		*Parameter*
	NoB	$N(b'_{1l})$	*NoB*	$N(b'_{2l})$	*NoB*	$N(b'_{3s})$	*NoB*	$N(b'_{4l})$	*r*
Variable	1	3	0	0	1	7	0	0	2
List	0	0	0	0	1	2	0	0	1
Constant	0	0	0	0	1	2	0	0	1
Structure	1	2	0	0	1	2	0	0	2

〈Table 3〉 The size of the maximum and the minimum indexing tree

Type of input key	Minimum Indexing Tree		Maximum Indexing Tree	
	Number of CpNodes	Number of LfNodes	Number of CpNodes	Number of LfNodes
Variable	1+2=3	12	1+2=3	12
List	1+1=2	4-1+2=5	1+1=2	4-1+2=5
Constant	1	4	1+1=2	4-1+2=5
Structure	1	4	1+2=3	4-2+2+2=6

As discussed earlier, the result shows that the maximum and the minimum indexing tree are the same if the input argument is either a variable or a list. In general terms, (1) up to $m+1$ choice points are needed under the WAM indexing if each group has at least one bucket that has more than one clauses. In addition, (2) two choice points are sometimes created contagiously in a given search path.

In order to verify the analysis result, we apply the result to predicate *match/2* in Fig. 1. As for the predicate, the number of clauses (n) is *12* and the number of partitions (m) is 4. In the predicate, for constant *a*, bucket $\{c_5, c_6\}$ is defined in group G_3. For structure *sum/2*, bucket $\{c_1, c_2\}$ and bucket $\{c_8, c_9\}$ are defined respectively in G_1 and G_3. Finally, bucket $\{c_{10}, c_{11}\}$ in group G_3 is defined for a list. As for a variable, buckets b_{1v} and b_{3v} have more than one clauses. The maximum number of buckets (the maximum value for r) that have more than one clauses can be selected is thus *2*, *1*, *1*, and *2* respectively for the variable, list, constant, and structure. The other parameters associated with each bucket are listed in Table 2. By using the parameters, p and r, as well as the number of clauses in buckets, we calculate the size of the indexing tree respectively for each data type. Its result is shown in Table 3. According to the table, up to three choice points are required in the execution of predicate *match/2* when the input key is either a variable or structure *sum/2*.

4. Flat Indexing

The analysis result in the previous section shows that up to $m+1$ choice points are needed when a predicate has m groups. It also shows that up to *2* contagious choice points may sometimes be created in a search path. The latter case implies that, for a given predicate, the amount of OR-parallelism (*i.e.*, the total number of clauses defining the predicate) may spread over the two choice points. This incurs a harmful influence on the parallelism exposition by decreasing the average amount of OR-parallelism per node. This section presents the Flat indexing scheme proposed to increase the average OR-parallelism per choice point.

4.1 Description of the Flat indexing

An input key may match with the two classes of clause keys, *i.e.*, either a variable

or the one of the same data type. In the WAM indexing scheme, each class is dealt with differently. Clauses with a variable key is defined as an independent group such that they are always tried. On the other hand, each non-variable group is divided into partitions according to the types of the clause key and thus only a single partition will be selected depending on the type of the input key. The advantage of this approach is that very compact code can be obtained since the WAM code for a predicate contains only a single copy of the code for the clauses with the variable key. However, as this way of grouping results in the division of the clauses to try into two kinds of disjoint sets, *i.e.*, the variable group and the partition chosen in the non-variable group according to the data type of the input key and as each kind may create respectively a choice point, the problem of the two choice points is raised.

The Flat indexing is motivated to solve the problem of these two choice points. In order to make only one choice point be created in a search path, the indexing scheme must ensure that, for a given input key, all the clauses to try must be contained in a *single* set. In the Flat indexing, all the clauses making a predicate are thus looked upon as *one* group whose meaning is basically the same as the one discussed in the WAM indexing. The group is divided into a set of partitions, named as P_v, P_l, P_c, and P_s as does under the WAM indexing. But different from the WAM indexing, partition P_x is composed of those clauses whose key is either a variable or a data of type x; the partition contains all the clauses subject to clause trials. For example of *match/2*, P_l is $\{c_4, c_{10}, c_{11}, c_{12}\}$ where the key of clauses c_{10} and c_{11} is a list and the key of clauses c_4 and c_{12} is a variable.

By defining the partition in the above way, for any input key, all the clauses to try are contained in a set (*e.g.*, partition). In the WAM indexing, when an input key fails to match with any clause with a non-variable key in the partition, the switching instructions dispatch the control to a failure service routine which will then lead to the processing of the next group. By doing this, the clauses with a variable key are always tried. Now that only a group exists in the Flat indexing, this case is treated differently. In addition to the normal partitions, a specific partition, called the *failure partition* P_f, is introduced which contains all the clauses with variable keys. For example of *match/2*, the failure partition becomes $\{c_4, c_{12}\}$. For a given key which does not match with any of clauses with non-variable keys, the switching instruction moves the control to the failure partition; thereby, all the clauses with variable keys are tried.

〈Table 4〉 The analysis result; b'c(resp. b's) is the bucket whose clause size is the largest among the buckets with a constant (resp. a structure) key.

Type of inpuyt key	*Maximum Number of CpNodes*	*Maximum Number of LfNodes*	*Minimum Number of LfNodes*
Variable	1	n	n
List	1	$N(b_l)$	b_l
Constant	1	$N(b'_c)$	b_l
Struct	1	$N(b'_s)$	b_l

〈Table 5〉 The maximum and minimum indexing tree for match/2 under the Flat indexing, where the numbers enclosed in parentheses are for the WAM indexing scheme

Type of input key	*Minimum Indexing Tree*		*Maximum Indexing Tree*	
	Minimun of CpNodes	*Number of LfNodes*	*Number of CpNodes*	*Number of LfNodes*
Variable	1(1)	12(12)	1(1)	12(12)
List	1(1)	4(5)	1(2)	4(5)
Constant	1(1)	2(4)	1(2)	5(5)
Structure	1(3)	2(4)	1(3)	6(6)

The implementation of the Flat indexing is made by using switching instructions as well. In the beginning of the code, instruction *switch_on_term* dispatches the control to a partition according to the data type of the first input argument. The semantics of the instruction is the same as the one in the WAM indexing except that the destination for each data type becomes the failure partition if there exists no clause whose key matches with the input key. If the partition selected by instruction *switch_on_term* is either P_c or P_s, instruction *switch_on_constant* or *switch_on_structure* dispatches the control to the appropriate subpartition depending on its data value of the input key. Again, these instructions are the same as those in the WAM indexing except that they have an additional pointer to the failure partition. If no bucket exists for the data value, the control is transferred to the failure partition as well. As does under the WAM indexing, partition P_v or P_l as well as subpartitions chosen with respect to constant or structure keys are called buckets and its code are organized by instructions *try*, *retry*, and *trust*. Fig. 6 depicts the code structure produced for a predicate under the Flat indexing. While the code is very similar to the one produced for a partition by the WAM indexing, it should be noted that arguments of instructions *switch_on_term*, *switch_on_constant*, and *switch_on_structure* are slightly different from those in the WAM indexing. In order to clearly show how a predicate is compiled under the Flat indexing, we provide in Fig. 7 the code structure for predicate *match/2*.

4.2 Analysis of the Flat Indexing

As we did for the WAM indexing, we derive the number of the *CpNodes* and the *LfNodes* in the indexing tree created for a predicate under the Flat indexing scheme. According to Table 4 which shows the result, the number of *CpNodes* created for a predicate is always one.

Table 5 shows the values obtained by applying the analysis result in Table 4 to predicate *match/2*. It also shows the values taken from Table 3. The results clearly show the difference between the WAM indexing and the Flat indexing: the number of choice points created for each predicate is always one in the Flat indexing, while it can be more than one in the WAM indexing. Interpreted within the context of OR-parallelism, the reduction of choice points means the increase of the amount of OR-parallelism exposed for each choice point. In other words, the Flat indexing contributes to the reduction of the non-leaf nodes in the runtime search tree; thereby, it increases the amount of OR-parallelism per node.

A close examination of *LfNodes* in the table reveals a strange result. that, the number of *LfNodes* between the WAM indexing and the flat indexing is sometimes different. For example, when the input is a list key, the number of *LfNodes* is four under the Flat indexing, but five under the WAM indexing. This is explained by using their indexing trees created when the input key is a list. Among the 5 *LfNodes* under the WAM indexing, it is found that the first *LfNode* is for the failure (*i.e.*, *SfNode* of case 1 resulting from *switch_on_term* instruction in the first group and the remaining ones are for clause tries (*i.e.*, *CtNodes* of case 1 or 2 in section 2) respectively for c_4, c_{10}, c_{11}, and c_{12}. On the other hand, all the four nodes under the Flat indexing scheme are for clauses tries (*i.e.*, *CtNodes* of case 1 or 2). As a matter of fact, leaf nodes in the Flat indexing consist always of *CtNodes*, whereas the leaf nodes in the WAM indexing consists of *CtNodes* as well as

```
Predicate:
                Switch_on_term
                [Start.C_partition | Ci | Failure_partition
Fail_partition  [Code for failure partition]
C_partition:    switch_on_constant
                [pointers to subpartitions, Fail_partition]
                [lists of subpartitions for constants]
L_partition:    [the bucket for lists]
S_partition:    Switch_on_structure
                [pointers to subpartitions, Fail_partition]
                [lists of subpartitions for structures]
Start:          [Code for clauses]
```

(Fig. 6) The structure of a predicate code

```
match_2:            swich_on_term C1_Label, C_partition,
L_partition
Fail_partition:     try C4_Code
                    tust C12_Code

C_partition:        switch_on_constant
                    (a: Ca_subpartition, b: Cb_subpartition)
Ca_subpartition:    try C4_Code
                    retry C5_Code
                    retry C6_Code
                    trust C12_Code
Cb_subpartition:    try C4_Code
                    retry C7_Code
                    trust C12_Code
L_partition:        try C4_Code
                    retry C10_Code
                    retry C11_Code
                    trust C12_Code
S_partition:        switch_on_structure 1, (Ssum-subpartition)
Ssum-subpartition:  try C1_Code
                    retry C2_Code
                    retry C4_Code
                    retry C8_Code
                    retry C9_Code
                    trust C12_Code
C1_Label:           try_me_else C2_Lael
C1_Code:            [Code for match(sum(A,B), sum(C,D)) :-
                    match(sub(A+D-1),sum(C+B-1).]
                    ; Codes for C3 - C11
C12_Label:          trust_me_esle_fail
                    [Code for match(X,numeric(Y)) :-
                    match(numeric(X), numeric(Y)).]
```

(Fig. 7) The code structure for match/2 under the Flat indexing

SfNodes. As a result, given an input key, the number of *LfNodes* is always smaller than or equal to the one in the WAM indexing. Interpreted within the context of the parallel execution, the removal of leaf nodes caused by the failure of switching instructions (case 3 in page 5) corresponds to the reduction of task switching by a scheduler [4,5]. In general parallel logic programming systems, the task switching is a very expensive operation because the scheduler must prepare the environment for the destination node[6,10]. When a scheduler performs task switching to a leaf node of case 3, it will finish the task right after the task switching, just wasting expensive system resource. Therefore, the reduction of leaf nodes caused by case 3 enhances the performance of parallel logic program systems by eliminating unnecessary task switching.

5. Experiment Results

In previous sections, we show that the size of the indexing tree generated under the Flat indexing is always smaller than that in the WAM indexing thanks to the reduction of the amount of *CpNodes* and *SfNodes*. The reduction of the number of choice points results is the primary benefit by increasing the amount of average OR-parallelism per choice point. Moreover, the removal of some terminal nodes caused by failure from instruction *switch_on_term* (*switch _on_constant*, or *switch_on_structure*) increases the parallel performance by reducing the total number of instructions to be executed as well as by eliminating the unnecessary scheduling activities with respect to the partition which will just fail in vain.

As opposed to the runtime benefits, the Flat indexing has a negative effect on the static code size. Primarily due to the failure partition, the code size becomes larger under

〈Table 6〉 The comparison of the indexing tree size between the Flat and the WAM indexing

Prolog Program	Flat Indexing		WAM indexing		Comparison	
	CpNodes (f1)	LfNodes (f2)	CpNodes (w1)	LfNodes (w2)	CpNode (w1/f1)	LfNodes (w2/f2)
boyer*	79476	89157	282097	194437	1.57	2.18
browse*	274714	271400	278387	281873	1.01	1.04
cal	30019	22641	30019	22641	1.00	1.00
chat_parser*	32620	39539	35845	40354	1.10	1.02
crypt	81	222	81	222	1.00	1.00
ham	359736	359734	359736	359734	1.00	1.00
meta_qsort*	2725	3598	2725	4405	1.00	1.22
nand*	8142	8566	8142	8665	1.00	1.01
mrv	580	578	580	578	1.00	1.00
poly_10*	14039	12531	18975	30733	1.35	2.45
queens10*	533231	533217	634592	634578	1.19	1.19
reducer*	10433	15986	11904	15986	1.14	1.00
sdda*	568	709	568	744	1.100	1.05
sendmore	12071	26128	12071	26128	1.100	1.00
tak	63625	15916	63625	15916	1.100	1.00
tak_gvar	790	418	790	418	1.00	1.00
zebra	14498	17315	14498	17315	1.00	1.00
Average/Average*					1.08/1.15	1.19/1.35

the Flat indexing due to failure partition. In the presence of the above trade-off, it is required to verify the performance of the Flat indexing by answering the following questions:

- What fraction of Prolog programs benefits from the Flat indexing?
- How much reduction can be achieved by the Flat indexing to the size of the indexing tree for the benchmarks which benefit from the Flat indexing?
- How much increase does the Flat indexing cause to the size of the code?

The first and second questions are to see how much effective the Flat indexing will be for practical applications. The third question is to see how less compact the code will be under the Flat indexing. Experiments made to answer these questions are based on the *TC-Prolog* (Threaded C-code Prolog) system. *TC-Prolog* is a sequential Prolog engine implemented via C code translation[8,11]. Different from other purely sequential Prolog systems, it is developed mainly for use as a sequential engine of the parallel implementation of Prolog. The normal *TC-Prolog* compiler produces the extended WAM code in which the indexing part is based on the Flat indexing scheme. Linked with an emulation engine, the code has been executed on a *HP's SPP-1200* multiprocessor system[7]. While the model we used has 16 CPUs, each of which is a PA-RISC 1.1, and runs under the SPP-IX 3.1 operation system, the evaluation is performed on a single processor configuration because our main concern is to identify the characteristics of the indexing tree. In the experiment, we insert some instrumental code extracting information on the indexing tree while the sequential Prolog code is executed. In addition, by modifying *TC-Prolog*, we implement another version that supports the WAM indexing. To distinguish between the two, the normal *TC-Prolog* will be called *TC-Prolog-FI* (the Flat indexing version of the *TC-Prolog*) and the version supporting the WAM indexing will be called *TC-Prolog-WI* (the WAM indexing version of the *TC-Prolog*).

We selected 17 benchmarks which have been frequently used in the evaluation of Prolog systems[2,8,12]. Respectively for each version, we measured the following three performance value: (1) the size of the indexing tree, (2) the size of assembly source, object, and executable code, and finally (3) the sequential execution time.

Table 6 shows the size of the indexing tree for each benchmark. Among the 17 benchmark programs, 9 benchmarks marked by asterisks benefit from the Flat indexing. As for the entire benchmarks, the WAM indexing creates 8% more choice points and causes 19% more *switch_on_term*, *switch_on_constant* or

switch_on_structure failures than the Flat indexing. As for the set of benchmarks benefiting from the Flat indexing, the WAM indexing creates 15% more choice points and causes 35% more *switch_on_term*, *switch_on_constant*, or *switch_on_structure* failures than the Flat indexing.

〈Table 7〉 The code size and execution time measured by TC-Prolog(Flat indexing) and the comparison of the code size where each entry is the rate of *TC-Prolog-WI* over *TC-Prolog-FI(i.e., TC-Prolog-WI/TC-Prolog-FI)*

Prolog Program	Assembly code (Kbytes)	Object code size (Kbytes)	Executable code size (Kbytes)	Execution time (msec)	Assembly code ratio	Object code ratio	Executable code ratio
boyer*	283	63	266	1374	0.95	0.94	1.00
browse*	79	20	237	1662	0.94	0.86	1.00
cal	75	19	237	180	0.99	0.95	1.12
chat_parser*	794	182	356	333	0.93	0.91	0.98
crypt	59	15	237	13	1.00	0.88	0.98
ham	61	16	233	1875	1.00	0.89	1.00
meta_qsort*	71	18	238	21	0.94	0.84	1.00
nand*	431	95	299	60	0.97	0.95	1.00
nrev	41	11	233	277	1.00	0.92	1.00
poly_10*	71	18	238	109	0.97	0.90	1.00
queens10*	41	11	233	6018	0.95	0.84	1.00
reducer*	217	51	262	100	0.95	0.84	1.00
sdda*	141	34	250	6	0.99	0.94	1.00
sendmore	59	14	233	139	1.00	0.94	1.00
tak	20	6	229	298	1.05	0.87	1.00
tak_gvar	26	8	229	10	1.04	0.90	1.00
zebra	42	12	233	112	1.02	0.93	1.00
Average/Average*					0.98/0.95	0.90/0.90	1.00/1.00

Table 7 shows the code size and the execution time measured for *TC-Prolog-FI*. The compiler used for the evaluation is gcc version *2.6.3* and all the compilation is with the optimization level *-O2*. Since *TC-Prolog -FI* translates Prolog into C via the WAM (Warren Abstract Machine), when measuring the assembly code size, we use *-S* option.

The righthand side of Table 7 also shows the ratio of *TC-Prolog-WI* to *TC-Prolog-FI* in terms of the code size and the execution time. On average, the assembly code size and the object code size in the Flat indexing are respectively 2% and 10% larger than those in the WAM indexing, while the executable code sizes are mostly the same. As for the benchmarks affected by the Flat indexing, the assembly code size and object code size in the Flat indexing are respectively 5% and 10% larger than those in the WAM indexing. This indicates that the Flat indexing does not lose much in terms of code compactness.

6. Concluding Remarks

Indexing is a method that prunes away unnecessary inferences in the evaluation of logic programs. To find an optimal indexing is quite complex and also demands a large number of abstract instructions for its implementation. In the indexing scheme of the WAM, the first argument of a clause is used as a key for indexing. In terms of the trade-off between efficiency and simplicity, using the first argument as a key is quite a reasonable choice. However, an invocation of a predicate may sometimes result in the creation of two contiguous choice points in a search path. In this case, because the OR-parallelism is expressed over the two choice points, the indexing scheme of the WAM is not efficient in terms of parallelism exposition. It is argued that the problem can be solved by compiling Prolog programs via different indexing schemes, and particularly the Flat indexing that we present in this paper. The experimental results show that over one half of the benchmarks benefit from the Flat indexing such that, compared with the WAM indexing, the number of choice points is reduced by 15%. Moreover, the amount of failures during the execution of indexing instructions is reduced by 35%. We believe that these reduction will contribute to higher

parallel performance thanks to the increased amount of average parallelism per node as well as to the decreased amount of task switching.

References

[1] H. Ait-Kaci and A. Podelski, Toward a Meaning of LIFE, Journal of Logic Programming, Vol.16, 195-234, 1993.

[2] Ali and R. Karlsson. Full Prolog and scheduling OR-parallelism in Muse. International Journal of Parallel Programming, 19:445-475, 1990.

[3] Ali and R. Karlsson. The Muse Approach to OR-parallel Prolog. International Journal of Parallel Programming, 19:129-162, 1990.

[4] A. Calderwood and P. Szeredi. Scheduling OR-parallelism in Aurora - the Manchester Scheduler. In Proceedings of the sixth International Conference and Symposium on Logic Programming, pp.419-435, 1989.

[5] A. Ciepielewski. Scheduling in OR-Parallel Prolog Systems: Survey and Open Problems. International Journal of Parallel Programming, Vol.20, No.6, pp.421-451, 1991.

[6] J. Conery, Binding Environments for Parallel Logic Programs in Non-Shared Memory Multiprocessors, International Journal of Parallel Programming, Vol.17, No.2, pp.125-152, April, 1989.

[7] Exemplar Architecture, Convex Press, Richardson, Texas, 1993

[8] C. Diaz and D. Diaz. wamcc: Compiling Prolog to C. In Proceedings of the Joint International Conference and Symposium on Logic Programming. MIT Press, Dec. 1995.

[9] G. Gupta and M. Hermenegildo, ACE: And/Or-parallel Copying-based Execution of Logic Programs, In Proceedings ICLP91 Workshop on Parallel Execution of Logic Programs, 1991.

[10] G. Gupta and B. Jayaraman, Analysis of Or-parallel Execution Models, ACM Transactions On Programming Languages and Systems, Vol.15, No.5, pp.659-680, Sep. 1993.

[11] B. Haussman, Turbo Erlang: Approaching the Speed of C, Kluwer, editor Evan Tick and Giancarlo Succi, pp.119-135, 1993.

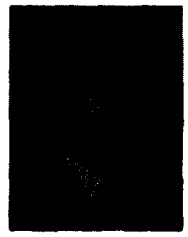
[12] F. Henderson and T. Conway and Z. Somogyi, Compiling logic programs to C using GNU C as a portable assembler, In Proc. of the JICSLP'95 Post conference on Implementation Techniques for Logic Programming Languages, Portland, USA, MIT Press, Dec., 1995.

[13] J. Jaffar and S. Michaylov and P. Stuckeyand R. Yap, The CLP(R) Languages and System, ACM Transactions on Programming Languages, Vol.14, No.10, pp.339-395, 1992.

[14] R. Kowalski, Logic for Problem Solving, Elsevier North-Holland, 1979.

[15] J. Lloyd. Foundation of Logic Programming, Spring-Verlag, 1987.

[16] D. Warren. An Abstract Prolog Instruction Set. Technical Report Technical Note 309, SRI, Oct. 1983.



김 희 철

1983년 연세대학교 전자공학과 졸업(공학사)

1983년~1988년 (주)삼성전자 주임연구원

1991년 Univ. of Southern Californi(Computer Eng. M.S.).

1996년 Univ. of Southern Californi (Computer Eng. Ph.D.)

1996년~1997년 (주)삼성SDS 수석연구원

1997년~현재 대구대학교 정보통신학부 전임 강사

관심분야 : 병렬처리, 컴퓨터구조, 컴파일러

이 용 두

1975년 한국항공대학교 통신학과 졸업(공학사)

1981년~1982년 (일)동경대학 전자공학과 객원 교수

1982년 영남대학교 대학원 전자공학과(공학석사)

1991년~1993년 Univ. of Southern California 교환교수

1995년 한국항공대학교 대학원 전자공학과(공학박사)

1995년~현재 대구대학교 정보통신공학부 교수

관심분야 : 컴퓨터구조, 컴퓨터통신, Internet 응용 기술