

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

HIECHEOL KIM, KANGWOO LEE⁺ AND JEAN-LUC GAUDIOT⁺⁺

*Department of Computer and Communication Engineering
Taegu University, Korea*

E-mail: hckim@biho.taegu.ac.kr

⁺*Department of Computer and Communication Engineering
Dongguk University, Korea*

E-mail: klee@cakra.dongguk.ac.kr

⁺⁺*Department of Electrical Engineering-Systems
University of Southern California*

Los Angeles, CA 90089-2563, U.S.A.

E-mail: gaudiot@usc.edu

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 the average parallelism per node (*i.e.*, *choice point*) as well as the number of clause trials. It also provides a way to evaluate the number of cases in which the control is passed to the failure code by means of an indexing instruction, such as *switch_on_term*, *switch_on_constant*, or *switch_on_structure*. By analyzing the indexing tree created when the indexing scheme is implemented in the WAM, we show the drawback of the WAM indexing scheme in terms of parallelism exposition and scheduling. Subsequently, we propose a new indexing scheme, which we call the *Flat indexing*. The experimental results show that over one half of our benchmarks benefit from 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 occur during the execution of indexing instructions is reduced by 35%.

Keywords: logic programming, clause indexing, OR-parallelism, WAM, Prolog

1. INTRODUCTION

Logic languages [17, 18] based on the SLD refutation [19] impose a strictly sequential search over the search tree. A technique called *indexing* is used to improve the performance when the number of clauses making up a predicate is large. Therefore, compilers for most logic languages produce code which implements a kind of such indexing [1,9].

In spite of research efforts over the last decade, the efficient implementation of Prolog is still an important issue in logic programming. As a sequential engine for logic languages, WAM (the Warren Abstract Machine) is a breakthrough which has contributed to highly efficient implementation of logic programs [16]. With 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 the sequential engine to achieve high single-thread performance [4, 12].

Received March 28, 1999; revised July 16, 1999; accepted November 26, 1999.
Communicated by Shang-Rong Tsai.

One choice point is that of runtime data structures created during execution of WAM code [1]. 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 for each invocation of a predicate, WAM sometimes creates two choice points which appear contagiously in a search path. However, these two choice points do not cause any significant disadvantage for the sequential WAM except for the cost of creating an additional choice point. Rather, they support very compact code, which is in fact one of the design objectives of WAM.

In parallel execution of WAM code, choice points have an additional role, that is, to represent OR-parallelism [3, 12]. This has caused the issue of 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, 12]. Because OR-parallelism can be crudely regarded as the number of alternative clauses for each predicate, the available OR-parallelism spreads over the two adjacent choice points, provided that two choice points are created. In this case, the available OR-parallelism is not fully exposed at the point of goal invocation; only a part of the OR-parallelism is available at the point of goal invocation. As a result, the creation of two choice points has a harmful effect on parallelism exposition by decreasing the average OR-parallelism.

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

Previously, a number of researches focused specifically on indexing [5, 6, 10, 14, 20]. Their main concern was fast resolution of clauses within the context of the sequential implementation of logic programs. To achieve fast resolution, some works introduced a formal model [14] while others used some auxiliary data structures, such as automata [20] and jump tables [10]. Even though they clearly supported fast resolution and unification, none of the previous works clearly addressed the multiple choice point problem, which flat indexing tries to solve.

The rest of this paper is organized as follows. Section 2 gives a brief introduction to the WAM indexing scheme. Section 3 presents a framework used for the analysis of indexing schemes and the results of WAM indexing obtained by applying the framework. Section 4 presents Flat indexing, which we propose to enhance OR-parallelism and scheduling efficiency. Section 5 reports the evaluation results. Finally, section 6 offers conclusions.

2. BACKGROUND

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

Given an input key (*i.e.*, the *first* argument of a goal), WAM indexing is applied to the predicate to prune out a subset of clauses before their clause trials, provided that these clauses will always fail in terms of unification. Among the four data types of WAM, (*variable*, *constant*, *list*, and *structure*), if a clause has a variable as its clause key (*i.e.*, the *first* argument of the clause), then the clause should not be pruned away because the clause key will always unite with an *input key* of any type. To make such clauses always subject to clause trials, WAM indexing divides the clauses, (c_1, \dots, c_n) which make up a predicate into a set of groups, G_1, \dots, G_m ($1 \leq m \leq n$), where G_i is a set of contiguous clauses of 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 the 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 $G_3 = \{c_5, c_6, c_7, c_8, c_9, c_{10}, c_{11}\}$, $G_4 = \{c_{12}\}$, where 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, the groups defined for a predicate are chained such that each group is visited consecutively at runtime regardless of the type of input key. Under such chaining, the clauses with variable keys are always tried. The implementation uses the instructions *try_me_else*, *retry_me_else* and *trust_me_else_fail* [1] as shown in Fig. 2.

c_1 :	<code>match(sum(A,B),sum(C,D)) :- match(sum(A+D-1), sum(C+B-1)).</code>
c_2 :	<code>match(sum(A,B),C) :- match(B-1, sum(C,B-1)).</code>
c_3 :	<code>match(a, b) :- match(numeric(a), numeric(b)).</code>
c_4 :	<code>match(X, ascii(Y)) :- match(ascii(X), digit(Y)).</code>
c_5 :	<code>match(a, b) :- match(ascii(a), ascii(b)).</code>
c_6 :	<code>match(a, b) :- match(digit(a), digit(b)).</code>
c_7 :	<code>match(b, X) :- match(digit(b), digit(X)).</code>
c_8 :	<code>match(sum(A,B), sum(C,D)) :- equal((A-C), equal(D-B)).</code>
c_9 :	<code>match(sum(A,B), C) :- match(sub(C-A), B).</code>
c_{10} :	<code>match([A,B],[C,D]) :- match(A,C), match(B,D).</code>
c_{11} :	<code>match([a,A], [C,b]) :- match(a,C), match(A,b).</code>
c_{12} :	<code>match(X, numeric(Y)) :- match(numeric(X), numeric(Y)).</code>

Fig. 1. Clauses making predicate *match/2*.

<i>match_2</i> :	<code>try_me_else G_{2_label}</code> <code>[Code for G₁]</code>
<i>G_{2_label}</i> :	<code>retry_me_else G_{3_label}</code> <code>[Code for G₂]</code>
<i>G_{3_label}</i> :	<code>retry_me_else G_{4_label}</code> <code>[Code for G₃]</code>
<i>G_{4_label}</i> :	<code>trust_me_else_fail</code> <code>[Code for G₄]</code>

Fig. 2. The structure of WAM code implementing the chaining of four groups in *match/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 tried. Actual indexing is applied to type β groups that consist of more than one clause. The indexing consists of two steps carried out firstly with respect to the *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 by 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 is 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. In the second step in indexing, one more level of partitioning is made for P_C , and P_S . In this step, the clauses having the same key value are grouped in a *subpartition*.

Given an input key of type x , the indexing conducted over the four partitions (P_C , P_L , P_S , P_V) selects one of the partitions, P_x , according to the type of input key. The control is then dispatched to partition P_x . In this way, the rest of the partitions can be excluded from clause trials. This implementation is conducted using the instruction *switch_on_term*, which has four arguments, respectively, for each type. The value of an argument has one of the following values.

- Case 1: the address of the clause's code (e.g., C_i) 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 clause.
- Case 3: the address of the code which processes the unification failure when no partition is defined for the given type.

```

Gi_code:
    switch_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 subpartitions}
                [Lists of subpartitions for structures]
Ci1:         try_me_else Ci2
                [Code for clause Ci1]
Ci2:         retry_me_else Ci3
                [Code for clause Ci2]
...
Cik:         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 $fail$.

When the partition thus selected using the 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 conducted using the 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. The entry holds the address of the subpartition's code if the subpartition contains more than one clause; otherwise, it holds either the address of the clause when there is a clause or the address of the failure code when there is no clause matching 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 or 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* and denoted as B_x where x is either l , v , c , or s to indicate the type of 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 clause. The WAM code of a bucket is organized such that all the clauses in the bucket are tried one by one. The implementation is conducted using the instructions *try*, *retry*, and *trust* [1]. Fig. 3 depicts the structure of the code for an example group with k clauses. Fig. 4 shows the code for group G_3 defined for *match/2*.

```

G3_code:      switch_on_term C5_Label, C_partition, L_partition, S_partition
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 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)).]
C7_Label:     retry_me_else C8_Label
C7_Code:     [Code for match(b,X) :- match(digit(b), digit(X)).]
C8_Label:     retry_me_else C9_Label
C8_Code:     [Code for match(sum(A,B), sum(C,D) :- equal((A-C),
                equal(D-B)).]
C9_Label:     retry_me_else C10_Label
C9_Code:     [Code for match(sum(A,B), C) :- match(sub(C-A), B).]
C10_Label:    retry_me_else C11_Label
C10_Code:    [Code for match([A,B], [C,D]) :- match(A,C), match(B,D).]
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 G_3 of *match/2*.

Before we proceed to the next section, we will briefly describe the relationship between WAM indexing and the number of choice points created for each invocation of a predicate. In WAM, the instruction *try_me_else* or *try* creates a choice point. When more than one clause is defined for a predicate, the code for the first clause always starts with the instruction *try_me_else*. In this case, if there is a bucket with more than one clause in any of these groups, its code starts with the instruction *try*. If the bucket is selected at runtime, two choice points are thus created contiguously in a search path, respectively, by means of *try_me_else* and *try*.

For example, the instruction `try_me_else` in the code for the 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 executed in parallel, respectively, by different processors. As depicted in the figure, if the input key is `a`, a new choice point is created inside group G_3 by the instruction `try` in C_a -subpartition (Fig. 4).

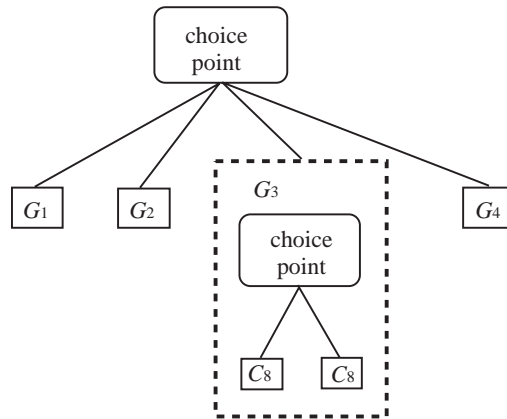


Fig. 5. Choice points and parallelism.

3. ANALYSIS FRAMEWORK FOR INDEXING SCHEMES

In the sequential execution of WAM code, the creation of two contiguous choice points for a predicate (*i.e.*, for execution of a goal) can be regarded just as a variation 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 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 WAM indexing and also a quantitative evaluation of the amount of OR-parallelism per choice point.

3.1 Analysis Framework: Indexing Tree

For a predicate P defined by more than one clause, we provide here some notations and definitions associated with WAM indexing. Let the set of clauses which make up predicate P be c_1, \dots, c_n . Suppose that the clauses are divided into m groups: G_1, G_2, \dots, G_m . Assume a mapping N such that $N(S)$ is 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.*, $\sum_{i=1}^m N(G_i) = n$). For a group G_i , let the buckets selected for each type of the key be b_{iv} , b_{is} , 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 defined for either a constant

or a structure key 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: $N(G_i) \geq N(b_{il}) + N(b_{ic}) + N(b_{is})$.

In order to analyze how many choice points are created in the execution of a goal, a tree 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, then 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 corresponding to a group. To understand the shape of a subtree, it must be noted that for each group, one of the following three cases occurs when *switch_on_term*, *switch_on_constant* or *switch_on_structure* is executed:

- 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 classified as a *SfNode* (*Switch failure Node*).
- Case 2: A bucket with only one clause is chosen; only one clause is tried. The subtree of this case is represented as a leaf node classified as a *CtNode* (*Clause Trial Node*).
- Case 3: A bucket b with more than one clause is chosen; more than one clause is tried. The subtree of this case consists of more than one node. The root of the subtree is always a *CpNode* because the number of clauses in the chosen bucket b is more than one. As $N(b)$ clauses will be tried with respect to this new *CpNode* and their trials will not cause any further 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, a non-leaf node is always a *CpNode* and a leaf node (*LfNode*) is always either a *SfNode* or a *CtNode*. Fig. 5 illustrates the form of an indexing tree for a predicate.

Given a predicate, its indexing tree may be defined differently depending on the data type of the input key. If the 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. For this case, to compare the sizes 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 that of *CpNodes* in T_2 .
- If T_1 and T_2 have the same number of *CpNodes*, then the one which has more leaf nodes is *larger* than the other.

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

- For a given data type, a maximum indexing tree of that type is any tree which is largest among the possible indexing trees.

≧ For a given data type, a minimum indexing tree of that type is any tree which is smallest among the indexing trees resulting from any input key of that type. When the input argument is either a constant or a structure, a minimum indexing tree is the one resulting from the input key which does not match any of the clause keys.

3.2 Results of the Analysis of WAM Indexing

In order to analyze WAM indexing, we use the framework established in the previous section. In the analysis, we calculate for each data type the size of a minimum and a 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. A minimum and a 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. Given an input key of the constant or the structure type, if the bucket chosen inside a group G_i consists of more than one clause, then we denote the bucket as b_i . For all i ($1 \leq i \leq m$), let r be the number of such buckets. Given an input key, the 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 a maximum indexing tree of the constant (*resp.* structure) type. In this case, we denote each of the buckets in group G_i ($1 \leq i \leq m$) as b'_{ic} (*resp.* b'_{is}).

Table 1 shows an analysis result, where p stands for the number of choice points created with respect to those groups. In other words, it becomes 0 if the number of 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 has been created for each of the buckets, the total number of *CpNodes* becomes p plus r .

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(s)		Maximum Indexing Tree(s)	
	Number of <i>CpNodes</i>	Number of <i>LfNodes</i>	Number of <i>CpNodes</i>	Number of <i>LfNodes</i>
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	$p+r$	$m-r + \sum_{i=1}^r N(b'_{is})$

The number of *LfNodes* in an indexing tree can be obtained by summing the number of *LfNodes* in 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 by using the input key.

As discussed earlier, the result shows that a maximum and a 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 WAM indexing if each group has at least one bucket that has more than one clause. 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 the predicate *match/2* shown 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 a constant a , the bucket $\{c_5, c_6\}$ is defined in group G_3 . For the structure *sum/2*, the bucket $\{c_1, c_2\}$ and the 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, the buckets b_{1v} and b_{3v} have more than one clause. The maximum number of buckets (the maximum value for r) that have more than one clause and can also 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. Using the parameters p and r as well as the number of clauses in buckets, we can calculate the size of the indexing tree, respectively, for each data type. The results are shown in Table 3. According to the table, up to three choice points are required in execution of the predicate *match/2* when the input key is either a variable or the structure *sum/2*.

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 r
	NoB	$N(b'_{ix})$	NoB	$N(b'_{ix})$	NoB	$N(b'_{3x})$	NoB	$N(b'_{ix})$	
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 sizes of the maximum and the minimum indexing trees.

Type of input key	Minimum Indexing Tree(s)		Maximum Indexing Tree(s)	
	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

4. FLAT INDEXING

The results of the analysis given in the previous section show that up to $m + 1$ choice points are needed when a predicate has m groups. They also show that up to 2 contiguous 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 has 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 mainly to increase the average OR-parallelism per choice point.

4.1 Description of the Flat Indexing

An input key may match 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 are 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 type of clause key; thus, only a single partition will be selected according to the type of 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 a choice point, the problem of two choice points occurs.

Flat indexing aims to solve the problem of these two choice points. In order to insure that only one choice point is created in a search path, the indexing scheme must ensure that, for a given input key, all the clauses to be tried must be contained in a *single* set. In Flat indexing, all the clauses making a predicate are, thus, looked upon as *one* group whose meaning is basically the same as that discussed for WAM indexing. The group is divided into a set of partitions, denoted as P_v , P_l , P_c , and P_s , as under WAM indexing. But different from WAM indexing, partition P_x , selected when the input key is of type x , is composed of those clauses whose key is either a variable or a data of type x ; the partition contains all the clauses which are subject to clause trials. 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 (Fig. 7).

Defining the partition in the above way, for any input key, the clauses to be tried are contained in a set (*e.g.*, partition). In WAM indexing, when an input key fails to match 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 processing of the next group. As a result, the clauses with a variable key are always tried. Now that only a group exists in 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 the example of *match/2*, the failure partition becomes $\{c_4, c_{12}\}$. For a given key which does not match any of the clauses with non-variable keys, the switching instruction moves the control to the failure partition; result, all the clauses with variable keys are tried.

The implementation of Flat indexing is also carried out using switching instructions. In the first part of the code, the 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 are the same as those in WAM indexing except that the destination for each data type becomes the failure partition if there exists no clause whose key matches the input key. If the partition selected by the instruction *switch_on_term* is either P_c or P_s , then the instruction *switch_on_constant* or *switch_on_structure* dispatches the control to the appropriate subpartition depending on the data value of the input key. Again, these instructions are the same as those in 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 under 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 is organized by

means of the instructions *try*, *retry*, and *trust*. Fig. 6 depicts the code structure produced for a predicate under Flat indexing. While the code is very similar to that produced for a partition by WAM indexing, it should be noted that the arguments of the instructions *switch_on_term*, *switch_on_constant*, and *switch_on_structure* are slightly different from those in 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 the predicate *match/2*.

<i>Predicate:</i>	<i>Switch_on_term</i> [<i>Start</i> , <i>C_partition</i> <i>C_i</i> <i>Failure_partition</i> , <i>L_partition</i> <i>C_i</i> <i>Failure_partition</i> , <i>S_partition</i> <i>C_i</i> <i>Failure_partition</i>]
<i>Failure_partition:</i>	[Code for <i>failure partition</i>]
<i>C_partition:</i>	<i>Switch_on_constant</i> [<i>pointers to subpartitions</i> , <i>Failure_partition</i>] [<i>lists of subpartitions for constants</i>]
<i>L_partition:</i>	[the bucket for the lists]
<i>S_partition:</i>	<i>Switch_on_structure</i> [<i>pointers to subpartitions</i> , <i>Failure_partition</i>] [<i>lists of subpartitions for structures</i>]
<i>Start:</i>	[Code for clauses]

Fig. 6. The structure of a predicate code.

<i>match_2:</i>	<i>switch_on_term</i> <i>C_{1_Label}</i> , <i>C_partition</i> , <i>L_partition</i> , <i>S_partition</i>
<i>Fail_partition:</i>	<i>try</i> <i>C_{4_Code}</i> <i>trust</i> <i>C_{12_Code}</i>
<i>C_partition:</i>	<i>switch_on_constant</i> 2, { <i>a</i> : <i>C_{a_subpartition}</i> , <i>b</i> : <i>C_{b_subpartition}</i> }
<i>C_{a_subpartition}:</i>	<i>try</i> <i>C_{4_Code}</i> <i>retry</i> <i>C_{5_Code}</i> <i>retry</i> <i>C_{6_Code}</i> <i>trust</i> <i>C_{12_Code}</i>
<i>C_{b_subpartition}:</i>	<i>try</i> <i>C_{4_Code}</i> <i>retry</i> <i>C_{7_Code}</i> <i>trust</i> <i>C_{12_Code}</i>
<i>L_partition:</i>	<i>try</i> <i>C_{4_Code}</i> <i>retry</i> <i>C_{10_Code}</i> <i>retry</i> <i>C_{11_Code}</i> <i>trust</i> <i>C_{12_Code}</i>
<i>S_partition :</i>	<i>switch_on_structure</i> 1, { <i>sum/2</i> : <i>S_{sum_subpartition}</i> }
<i>S_{sum_subpartition}:</i>	<i>try</i> <i>C_{1_Code}</i> <i>retry</i> <i>C_{2_Code}</i> <i>retry</i> <i>C_{4_Code}</i> <i>retry</i> <i>C_{8_Code}</i> <i>retry</i> <i>C_{9_Code}</i> <i>trust</i> <i>C_{12_Code}</i>
<i>C_{1_Label}:</i>	<i>try_me_else</i> <i>C_{2_Label}</i>
<i>C_{1_code}:</i>	[Code for <i>match(sum(A,B), sum(C,D))</i> : - <i>match(sub(A+D-1), sum(C+B-1))</i> .]; Codes for <i>C₂-C₁₀</i>
<i>C_{11_Label}:</i>	<i>retry_me_else</i> <i>C_{11_Label}</i>

C_{11_Code} : [Code for $match([a,A], [C,b]) :- match(a,C), match(A,b).$]
 C_{12_Label} : $trust_me_else_fail$
 C_{12_Code} : [Code for $match(X, numeric(Y)) :- match(numeric(X), numeric(Y)).$]

Fig. 7. A code structure for $match/2$ under Flat indexing.

4.2 Analysis of Flat Indexing

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

Table 4. The analysis results: 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 input 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_f
Structure	1	$N(b'_s)$	b_f

Table 5 shows the values obtained by applying the analysis results in Table 4 to the 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.

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	
	Number of $CpNodes$	Number of $LfNodes$	Number of $CpNodes$	Number of $LfNodes$
Variable	1(3)	12(12)	1(3)	12(12)
List	1(2)	4(5)	1(2)	4(5)
Constant	1(1)	2(4)	1(2)	4(5)
Structure	1(1)	2(4)	1(3)	6(6)

A close examination of the $LfNodes$ in the table reveals a strange result. Normally, the number of $LfNodes$ between the WAM indexing and the flat indexing is 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 five *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 clause 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 *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 section 2) corresponds to the reduction of task switching by a scheduler [4, 7]. 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 [8, 13]. 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 is the primary benefit by increasing the amount of average OR-parallelism per choice point. Moreover, the absence of some terminal nodes caused by failure from instruction *switch_on_term* (*switch_on_constant*, or *switch_on_structure*) improves the parallel performance by reducing the total number of instructions to be executed as well as by eliminating unnecessary scheduling activities with respect to the partition which will fail.

As opposed to these runtime benefits, Flat indexing has a negative effect on the static code size. Primarily due to the failure partition, the code size becomes larger under the Flat indexing. In the presence of the above trade-off, it is necessary to verify the performance of Flat indexing by answering the following questions:

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

The first and second questions aim to see how effective Flat indexing will be for practical applications. The third question aims to find out how compact the code will be under Flat indexing. Experiments conducted to answer these questions are based on the *TC-Prolog* (Thread-Code Prolog) system. The TC-Prolog is a sequential Prolog engine implemented via C code translation [11, 15]. Different from other purely sequential Prolog systems, it was developed mainly for use as a sequential engine in parallel implementation of Prolog.

The normal *TC-Prolog* compiler produces 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 [9]. 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 was performed on a single processor configuration because our main concern was to identify the characteristics of the indexing tree.

In the experiment, we inserted some instrumental code extracting information into the indexing tree while the sequential Prolog code was executed. In addition, by modifying the *TC-Prolog*, we could implement another version that supports WAM indexing. To distinguish between them, the normal *TC-Prolog* will here be called *TC-Prolog-FI* (the Flat indexing version of *TC-Prolog*) and the version supporting WAM indexing will be called *TC-Prolog-WI* (the WAM indexing version of *TC-Prolog*).

We selected 17 benchmarks which have been frequently used in the evaluation of Prolog systems[2, 11, 16]. For each version, we measured the following three performance criteria: (1) the size of the indexing tree, (2) the size of the assembly source, object, and executable code, and (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 with asterisks in the table benefited from Flat indexing. As for all of the benchmarks, WAM indexing created 8% more choice points and caused 19% more *switch_on_term*, *switch_on_constant* or *switch_on_structure* failures than did Flat indexing. As for the set of benchmarks benefiting from Flat indexing, WAM indexing creates 15% more choice points and caused 35% more *switch_on_term*, *switch_on_constant*, or *switch_on_structure* failures than did *Flat indexing*.

Table 6. A comparison of the indexing tree size between Flat and WAM indexing

Prolog Program	Flat Indexing		WAM indexing		Comparison	
	CpNodes (f1)	LfNodes (f2)	CpNodes (w1)	LfNodes (w2)	CpNode (w1/f1)	LfNode (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	1.22
meta_qsort*	2725	3598	2725	4405	1.00	1.22
nand*	8142	8566	8142	8665	1.00	1.01
nrev	580	578	580	578	1.00	1.00
poly_10*	14039	12531	18975	30733	1.35	2.45
queens 10*	533231	533217	634592	634578	1.19	1.19
reducer*	10432	15986	11904	15986	1.14	1.00
sdda*	568	709	568	744	1.00	1.05
sendomre	12071	26128	12071	26128	1.00	1.00
tak	63625	15916	63625	15916	1.00	1.00
tak_gvar	790	418	790	418	1.00	1.00
zebra	14498	17315	14498	17315	1.00	1.00

Table 7. The code size and execution time measured for TC-Prolog(Flat indexing), and a 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	Execution time ratio	
boyer*	283	63	266	1374	0.95	0.94	1.00	1.15	
browse*	79	20	237	1662	0.94	0.86	1.00	0.97	
cal	75	19	237	180	0.99	0.95	1.12	1.00	
chat_parser*	794	182	356	333	0.93	0.91	0.98	1.00	
crypt	59	15	237	13	1.00	0.88	0.98	1.15	
ham	61	16	233	1875	1.00	0.89	1.00	0.97	
meta_qsort*	71	18	238	21	0.94	0.84	1.00	1.14	
nand*	431	95	299	60	0.97	0.95	1.00	0.95	
nrev	41	11	233	277	1.00	0.92	1.00	0.96	
poly_10*	71	18	238	109	0.97	0.90	1.00	1.16	
queens 10*	41	11	233	6018	0.95	0.84	1.00	0.95	
reducer*	217	51	262	100	0.95	0.84	1.00	0.99	
sdda*	141	34	250	6	0.99	0.94	1.00	2.67	
sendmore	59	14	233	139	1.00	0.94	1.00	0.98	
tak	20	6	229	298	1.05	0.87	1.00	1.02	
tak_gvar	26	8	229	10	1.04	0.90	1.00	1.00	
zebra	42	12	233	112	1.02	0.93	1.00	0.97	
Average/Average*						0.98/ 0.95	0.90/ 0.90	1.00/ 1.00	1.10/ 1.22

Table 7 shows the code size and the execution time measured for *TC-Prolog-FI*. The compiler used for the evaluation was *gcc* version 2.6.3, and the compilation was all done at optimization level-*O2*. Since *TC-Prolog-FI* translates Prolog into C via WAM(Warren Abstract Machine), when measuring the assembly code size, we used the *-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 Flat indexing are, respectively, 2 and 10% larger than those in WAM indexing, while the executable code sizes are mostly the same. As for the benchmarks affected by Flat indexing, the assembly code size and object code size in Flat indexing are, respectively, 5 and 10% larger than those in WAM indexing. This indicates that 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 scheme is quite complex and also demands a large number of abstract instructions for the implementation. In the indexing scheme of 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 WAM is not efficient in terms of parallelism exposition. It has been argued that the problem can be solved by compiling Prolog programs via different indexing schemes, particularly Flat indexing, which we have presented in this paper. The experiments conducted to compare WAM indexing with Flat indexing show that over one half of the benchmarks benefit from Flat indexing such that, compared with WAM indexing, the number of choice points is reduced by 15%. Moreover, the number of failures during execution of indexing instructions is reduced by 35%. We believe that these reductions will contribute to better 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, *Warren's Abstract Machine: A Tutorial Reconstruction*, Logic Program Series, MIT Press, 1991.
2. Ali and R. Karlsson, "Full prolog and scheduling OR-parallelism in muse," *International Journal of Parallel Programming*, Vol. 19, No. 6, 1990, pp. 445-475.
3. Ali and R. Karlsson, "The muse approach to OR-parallel prolog," *International Journal of Parallel Programming*, Vol. 19, 1990, pp.129-162
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*, 1989, pp. 419-435
5. M. Carlsson, "Freeze, indexing, and other implementation issues in the WAM," in *Preceedings of International Conference on Logic Programming '87*, 1987, pp. 40-58.
6. R. Colomb, "Enhancing unification in Prolog through clause indexing," *Journal of Logic Programming*, Vol. 10, No. 1, 1991, pp. 23-44
7. A. Ciepielewski, "Scheduling in OR-parallel Prolog systems: survey and open problems," *International Journal of Parallel Programming*, Vol. 20, No. 6, 1991, pp. 421-451.
8. J. Conery, "Binding environments for parallel logic programs in non-shared memory multiprocessors," *International Journal of Parallel Programming*, Vol. 17, No. 2, 1989, pp. 125-152.
9. *Exemplar Architecture*, Convex Press, Richardson, Texas, 1993
10. S. Dawson, C. Ramakrishnan and I. Ramakrishnan, "Design and implementation of jump tables for fast indexing of logic programs," in *Preceedings of Symposium on Programming Language Implementation and Logic Programming '95*, 1995, pp. 98-105.
11. C. Diaz and D. Diaz, "WAMCC: compiling Prolog to C," in *Proceedings of the Joint International Conference and Symposium on Logic Programming*, MIT Press, 1995, pp. 341-351.
12. E. Pontelli, G. Gupta, and M. Hermenegildo, "&-ACE a high performace parallel prolog system," in *Proceedings of International Parallel Processing Symposium*, IEEE Press, 1995, pp. 564-571.
13. G. Gupta and B. Jayaraman, "Analysis of Or-parallel execution models," *ACM Transactions on Programming Languages and Systems*, Vol. 15, No. 5, 1993, pp. 659-680

14. W. Hans, "A complete indexing scheme for WAM-based abstract machine," in *Proceedings of Symposium on Programming Language Implementation and Logic Programming '92*, 1992, pp. 232-244.
15. B. Hausman, *Turbo Erlang: Approaching the Speed of C*, Kluwer, Evan Tick and Giancarlo Succi (ed.), 1993, pp. 119-135.
16. F. Henderson, T. Conway and Z. Somogyi, "Compiling logic programs to C using GNU C as a portable assembler," in *Proceedings of the JICSLP' 95 Post Conference on Implementation Techniques for Logic Programming Languages*, MIT Press, 1995, pp. 21-35.
17. J. Jaffar, S. Michaylov, P. Stuckey and R. Yap, "The CLP(R) languages and system," *ACM Transactions on Programming Languages*, Vol. 14, No. 10, 1992, pp. 339-395.
18. R. Kowalski, *Logic for Problem Solving*, Elsevier North-Holland, 1979
19. J. Lloyd, *Foundation of Logic Programming*, Springer-Verlag, 1987
20. R. Ramesh, I. Ramakrishnan and D. S. Warren, "Automata-driven indexing of Prolog clauses," in *Proceeding of Symposium on Principles of Programming Language '90*, 1990, pp. 281-291.



Hiecheol Kim received his B.S. degree in Electronic Engineering from Yousei University, Korea and his M.S. and Ph.D. degrees in Computer Engineering from University of Southern California in 1991 and 1996, respectively. From 1983 to 1988 he was a researcher in Samsung Electronics, Ltd., Korea. Since 1997, he has been on a faculty of the department of Computer and Communication Engineering, Taegu University, Korea, where he is currently an assistant professor. His current research interests include parallel compiler, parallel processing, parallel architecture, and reconfigurable computing.



Kangwoo Lee received his B.S. degree in Electronic Engineering from Yousei University, Korea and his M.S. and Ph.D. degrees in Computer Engineering from Syracuse University and University of Southern California in 1993 and 1998, respectively. Since graduating in 1999, he has been on a faculty of the department of Computer and Communication Engineering, Dogguk University, Korea, where he is currently an assistant professor. His current research interests include parallel processing, parallel architecture, and fault-tolerant computing.



Jean-Luc Gaudiot received the Diplome d'Ingenieur from the Ecole Superieure d'Ingenieurs en Electrotechnique et Electronique, Paris, France in 1976 and the M.S. and Ph.D. degrees in Computer Science from the University of California, Los Angeles in 1977 and 1982, respectively. Since graduating in 1982, he has been on the faculty of the Department of Electrical Engineering-Systems, University of Southern California, where he is currently a Professor. His research interests include multithreaded architectures, fault-tolerant multiprocessors, and implementation of artificial neural networks. He is the Program Committee chair of the High Performance Computer Architecture Conference in 1999 (HPCA-4) and is currently the Editor-in-Chief of the IEEE Transactions on Computers and as Advisory Board member of the IEEE Technical Committee on Computer Architecture. He is a Fellow of the IEEE.