A Method of Vector Processing for Shared Symbolic Data

Yasusi Kanada

Central Research Laboratory, Hitachi Ltd.
P. O. Box 2, Kokubunji, Tokyo 185, Japan.
kanada@crl.hitachi.co.jp

Abstract

The conventional processing techniques for pipelined vector processors such as Cray-XMP, or SIMD parallel processors, such as CM-2 (connection machine), are generally applied only to independent multiple data processing. This paper describes a vector processing method of multiple processings including parallel rewriting of dynamic data structures with shared elements, and of multiple processings that may rewrite the same data element two or more times. This method is called the filtering-overwritten-label method (FOL). FOL enables vector processing of entering multiple data into a hash table, address calculation sorting, and many other algorithms that handle lists, trees, graphs and other types of symbolic data structures. FOL is applied to several symbolic processing algorithms; consequently, the performance is improved by a factor of ten on the Hitachi S-810.

1. Introduction

An attached vector processor, called the Hitachi M-680H IDP (Integrated Database Processor) [Koj 87], is designed for database processing and has been applied to several symbolic processing applications [Tor 88]. However, most other vector processors, such as the Cray-XMP or the Hitachi S-820, are mostly used for numerical processing, and rarely used for symbolic processing. One of the reasons that the extension of applications from numerical to non-numerical areas has been prevented is that no vectorization method that is widely applicable to processing dynamic data structures connected by pointers, such as linear lists, trees and graphs has yet been established.

The symbolic vector-processing methods developed by Kanada, et. al. [Kan 88, Kan 89a, Kan 89b] enable vector processing of multiple dynamic data structures by *vectorization*, a program transformation. When these methods are applied, the data structures are accessed through *index vectors*, which contain pointers or indices to the data to be processed. These methods are called *simple index-vector-based vector-processing methods* (SIVP) in this paper. The list-vector-processing facility and conditional control facilities [Kam 83], such as masked operations, of vector processors are used in SIVP.

However, the conventional vector-processing methods including SIVP are basically applied only to *independent* multiple data processings. That means these methods cannot vectorize multiple processings including rewriting of data with shared elements, such as graphs, and they cannot vectorize multiple processings that may rewrite the same data element two or more times, such as entering multiple data into a hash table, as will be explained in Section 2. The *filtering-overwritten-label method* (FOL) explained in this paper solves this problem.

The above problem is explained further in Section 2. The principle and algorithms of FOL are shown in Section 3. Several applications and performance evaluations of FOL are shown in Section 4. The related works are mentioned briefly in Section 5.

2. Problems in Vector Processing of Shared Data

In the symbolic vector-processing methods shown in Kanada [Kan 88, Kan 89a, Kan 89b], data are read and written through index vectors. **Figure 1** illustrates two types of index vectors: a vector of pointers to the data, and a vector of subscripts or displacements of the data. Using index vectors, parts of the symbolic data are gathered into a vector register or scattered to the main storage by the so-called list-vector instructions or indirect vector load/store instructions.

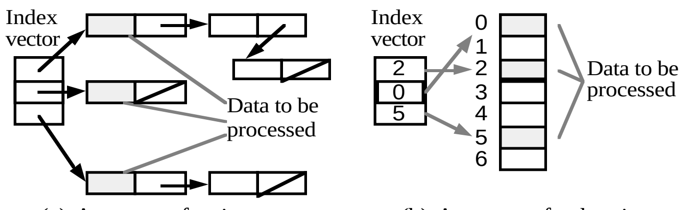


Figure 1. Two types of index vectors

The classes of vectorizable and unvectorizable processings are explained using **Figure 2**. SIVP is basically applicable only for independent multiple data processings. That means SIVP can be applied to an index vector that contains pointers or indices to independent data (Figure 2a). However, SIVP can also be applied to *read-only* processings of multiple data including shared data. The index vector may have pointers to the same data (Figure 2b), as long as it does not update the data. But SIVP cannot be applied for *rewriting* multiple data with sharing (Figure 2b), because if applied, processings that must be performed sequentially would be performed in parallel and would cause incorrect results (explained later in this section). Therefore, SIVP cannot be applied for rewriting partially shared data structures (illustrated in **Figure 3**), because read-only vector-processable index vectors (Figure 2b) may be generated while processing such data structures.

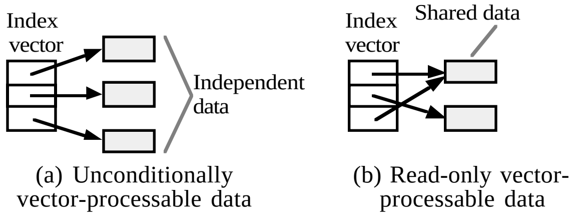

Index vector
Independent data
Index vector
Shared data
(a) Unconditionally vector-processable data
(b) Read-only vector-processable data

Figure 2. Vector-processable data for the previous methods

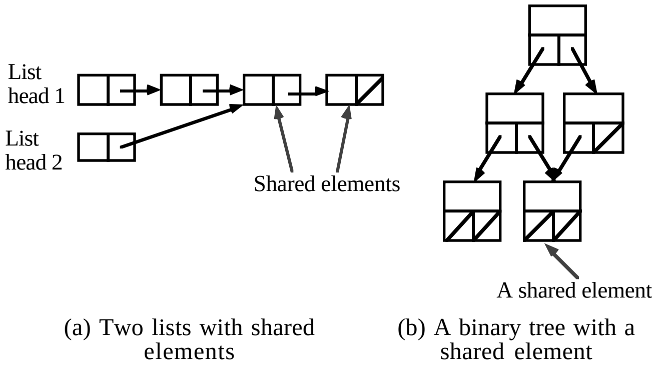

List head 1
List head 2
Shared elements
A shared element
(a) Two lists with shared elements
(b) A binary tree with a shared element

Figure 3. Examples of partially shared data structures

Two examples that cannot be processed by the previous SIVP methods are shown below (in **Figure 4**). The first example shows multiple data being entered into a hash table. This processing is called *multiple hashing* in this paper. The entered data are chained from the hash table entries. Figure 4a shows sequential processing whereby two keys are being entered. Keys 353 and 911 are entered in this order. Because the hashed values of these keys are both five, they are colliding. So they are chained from the same hash table entry. Figure 4b shows the problem of multiple hashing by *forced* vector processing. The keys are initially stored in a vector, and the hashed values are calculated by vector operations and stored into another vector which is used as an index vector in the following process. If no collisions occur, the key writing is processed properly. However, collision occurrence makes the correct processing impossible. The pointer to the second key overwrites that to the first key in this figure.

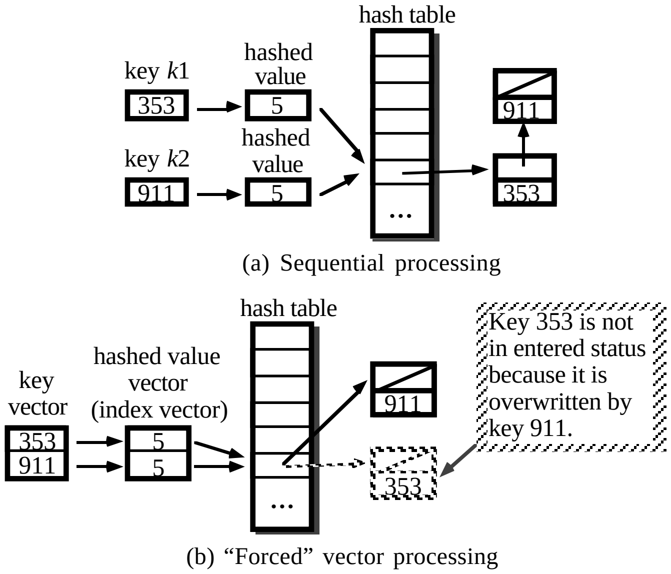

hash table
key k1
353
hashed value
5
key k2
911
hashed value
5
...
911
353
(a) Sequential processing
hash table
key vector
353
911
hashed value vector (index vector)
5
5
...
911
353
Key 353 is not in entered status because it is overwritten by key 911.
(b) "Forced" vector processing

Figure 4. The problem of multiple hashing by vector processing

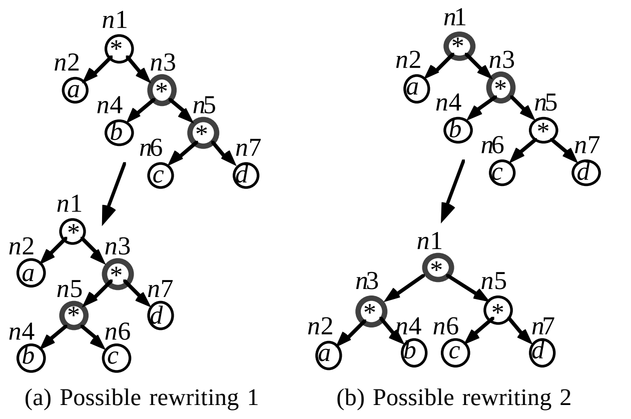


Figure 5. Two ways of rewriting a tree

The second example is tree rewriting which transforms an input tree to an equivalent final form by applying a rewriting rule. **Figure 5** illustrates two ways of rewriting an operation tree; here, the associative law is used as the rewriting rule. The associative law is expressed as $X * (Y * Z) \rightarrow (X * Y) * Z$. The arrow indicates the direction of the rewriting. The input tree is $a * (b * (c * d))$ in Figure 5. The rewriting is applied to nodes $n3$ and $n5$ in Figure 5a, and to nodes $n1$ and $n3$ in Figure 5b. Node $n3$ is "shared" between these two rewritings. A *forced* parallel rewriting by vector processing causes generation of a tree which is not equivalent to the original, or the rewriting process will be aborted because of a nonexistent (phantom) node access. In this example, multiple (two) nodes are rewritten in a unit process, i.e., one rewriting.

3. A Solution: the Overwritten-Label Filtering Method

A method of rewriting multiple symbolic data with sharing, called the *filtering-overwritten-label method* (FOL), has been developed. The principle and algorithms of FOL are explained in this section.

3.1 The principle of FOL

The problem explained in Section 2 is solved by the decomposition of a data set into parallel-processable subsets, illustrated in **Figure 6**. In this principal method, the element set (S) of the original index vector is split into parallel-processable sets ($S1$, $S2$, and $S3$) and restored into parallel-processable index vectors ($V1$, $V2$, and $V3$). These index vectors are processed one-by-one by vector operations. Pointers to one data, *a* for example, are scattered into different index vectors. This principal method processes the *unshared* part of the multiple data in parallel and the shared part sequentially.

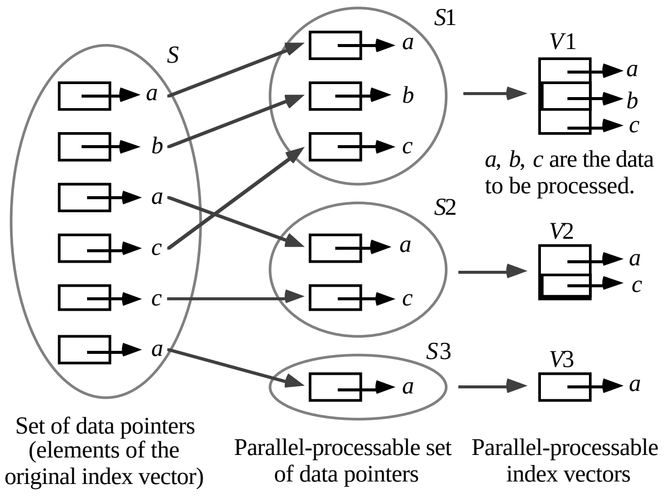

S
S1
S2
S3
V1
V2
V3
a
b
c
a, b, c are the data to be processed.
Set of data pointers (elements of the original index vector)
Parallel-processable set of data pointers
Parallel-processable index vectors

Figure 6. Decomposition of data with sharing into parallel-processable index vectors

To implement the above principal method, a method of decomposing data sets into parallel-processable sets must be developed. The data which cannot be processed in parallel are multiple pointers or indices that point to *one* data, and they can be detected by comparing all of the pairs of pointers or indices. However, this process needs $O(N^2)$ comparisons, so it will decrease the performance. FOL[*] is a method of decomposing a data set in $O(N)$ time in normal cases by vector processing.

Before explaining the algorithms of FOL in the following subsections, a method of multiple hashing using FOL is explained informally (See **Figure 7**). Only the keys are entered into the hash table in this example for the sake of simplicity. The keys to be entered are initially stored in a vector. Each hash table entry has a work area for storing *labels*.

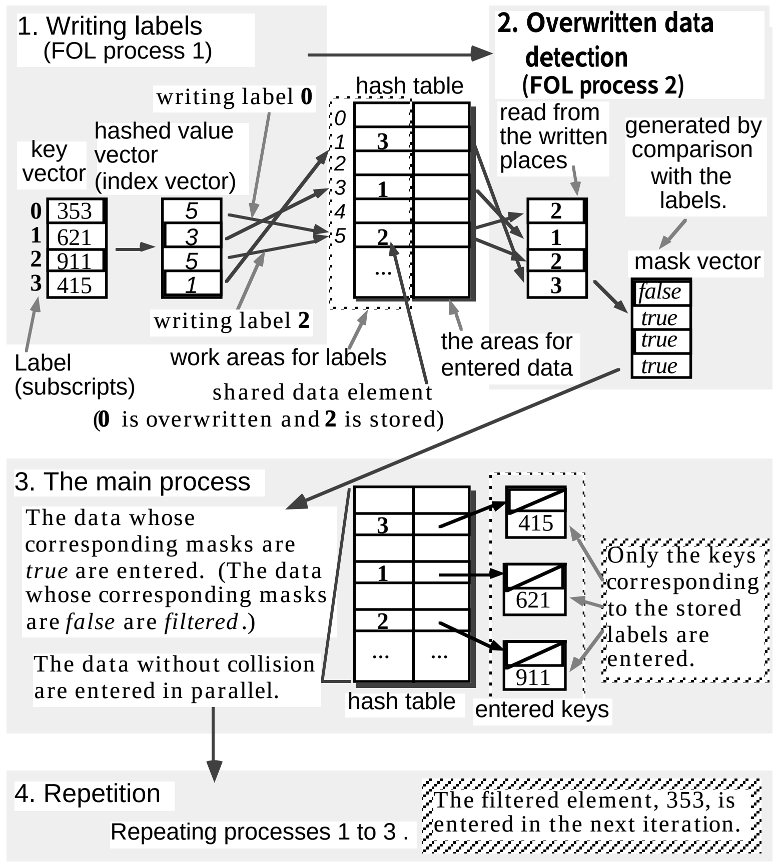
The detection of collisions, entitled FOL processes 1 and 2 in Figure 7, is performed entirely by vector operations in the following manner. In Step 1, the subscripts of the key vector are written into the work area indexed by the hashed values. These subscripts are called *labels* in FOL. In Step 2, the labels are read immediately after writing, using the same indices, i.e., the hashed values. These label writing and reading processes are performed using the list-vector instructions. The elements of the read vector are compared with the original labels. They are equal if there are no collisions. However, they are not equal when there are collisions, because collisions cause overwriting of labels in the work areas. The results of the comparisons are stored into a mask vector, a Boolean vector. The key vector elements, whose corresponding mask vector elements are *true*, are the set of parallel-processable data. In the example in Figure 7, the second to fourth elements of the key vector form the first parallel-processable set of keys, because the second to fourth elements of the mask vector are *true*. These keys are entered into the hash table in parallel in Step 3. In Step 4, the above process is repeated until all of the keys are classified into a parallel-processable set, and all of them are entered into the hash table.

3.2 FOL for rewriting single data per unit process

FOL is a generalization of the multiple hashing method explained in the previous subsection. FOL can be applied to a wide range of processings that rewrite multiple data with possible sharing. The algorithm of FOL for rewriting a single data per unit process to be vectorized is shown in this subsection. An extension of FOL for rewriting multiple data per unit process, such as rewriting the operation tree shown in Section 2, is shown in the next subsection.

An FOL algorithm which decomposes a set of data into parallel-processable sets is shown below. The whole process of this algorithm can be performed by vector operations on a vector processor such as the Hitachi S-820.

[*] FOL is a generalization of the "overwrite-and-check" method in Kanada [Kan 90a].



1. Writing labels (FOL process 1)
2. Overwritten data detection (FOL process 2)
writing label 0
hash table
read from the written places
generated by comparison with the labels.
key vector
hashed value vector (index vector)
mask vector
0 353
1 621
2 911
3 415
5
3
5
1
false
true
true
true
writing label 2
work areas for labels
the areas for entered data
Label (subscripts)
shared data element (0 is overwritten and 2 is stored)
3. The main process
The data whose corresponding masks are true are entered. (The data whose corresponding masks are false are filtered.)
The data without collision are entered in parallel.
Only the keys corresponding to the stored labels are entered.
hash table
entered keys
4. Repetition
Repeating processes 1 to 3 .
The filtered element, 353, is entered in the next iteration.

Figure 7. A method of multiple hashing by FOL

■ Algorithm: The Overwritten-Label Filtering Method 1 (FOL1)

❏ Input

This algorithm inputs an index vector V. The elements of V are pointers or indices to storage areas containing data $d_1, d_2, \ldots, d_N$, respectively, where there may be duplicated data (i.e., the same data may appear two or more times in the sequence $d_1, d_2, \ldots, d_N$). The storage area pointed to by v, an element of V, is noted by $v\rightarrow$, and the data stored in $v\rightarrow$ is noted by $v\rightarrow d$.

❏ Output

This algorithm outputs the sets of parallel-processable data $S_1, S_2, \ldots, S_M$ (the value of M is obtained as an execution result of this algorithm), where $\bigcup_{i=1}^{M} S_i = \{d_1, d_2, \ldots, d_N\}$ and $S_1, S_2, \ldots, S_M$ are disjoint sets, i.e., $S_i \cap S_j = \varnothing$ for arbitrary i and j (***disjoint decomposition condition***).

❏ Process conditions

- Processing P_i, which corresponds to the main process in Figure 7, is applied to data d_i ($i = 1, 2, \ldots, N$) by vector operation after the execution of this algorithm. All the data included in an output set S_j ($j = 1, 2, \ldots, M$) may be processed in parallel or in arbitrary order in processing P_i.[*] The order of processing for each output set can be in arbitrary order, but two data belonging to different output sets may not be executed in parallel. So any two of the output sets must be processed sequentially. The execution order between the processings of two arbitrary data must not affect the correctness of the result.
- If va and vb are arbitrary elements of V before the execution, $va\rightarrow$ and $vb\rightarrow$ are possibly the same storage area. However, the data pointed to by va and that pointed to by vb are regarded as non-identical data in the algorithm description.
- For each element v of the index vector V, a work area noted by $v\rightarrow w$ is allocated in the storage area $v\rightarrow$. This means that, if $va\rightarrow$ and $vb\rightarrow$ are the same storage area, $va\rightarrow w$ and $vb\rightarrow w$ are also the same work area even when va and vb are not identical.

❏ Procedure

(0) Preprocessing: Set 1 to variable j. Assign a unique label to each element of V. The labels may be assigned before the execution time if possible.[**]

(1) Writing labels: Write the labels of $v_1, v_2, \ldots, v_n$ into the work areas $v_1\rightarrow w, v_2\rightarrow w, \ldots, v_n\rightarrow w$, respectively; where n is the number of elements of V. The execution order is arbitrary, and the labels may be processed in parallel.

(2) Detection of overwriting: Read the labels from the work areas $v_1\rightarrow w, v_2\rightarrow w, \ldots, v_n\rightarrow w$, and compare them with the labels of $v_1, v_2, \ldots, v_n$, respectively. Step (1) must be completed before reading the labels. (A synchronization may be necessary.) If the value read from $v_i\rightarrow w$ is not equal to the label of v_i, it means that v_i is overwritten. Then the set of all the data pointed to by the elements of V except the data whose equality check has failed is the parallel-

[*] This condition holds for multiple hashing. Because if no collision occurs, the order does not matter. And if a collision occurs, the order of the entered data in the hash table is dependent on which colliding data is stored, but again it does not matter. Processings not satisfying this condition are considered later in a footnote.

[**] The most easily computable label for element v in vector V is the index or element number of v in V, or the displacement of v (the number of bytes) from the top address of V.

processable set. This set is named S_j. More exactly, set $\{u_1 \rightarrow d, u_2 \rightarrow d, \ldots, u_m \rightarrow d\}$ is assigned to S_j, where u_k ($k = 1, 2, \ldots, m$) are all the elements of V, which satisfies the relation $u_k \rightarrow w = l_k$; where l_k is the label for u_k.

(3) Updating control variables: Add 1 to j. Delete the pointers or indices pointing to data in S_j, from V. The number of elements of V is reduced.

(4) Repetition: Repeat the above Steps (1) to (3) until V becomes empty. When terminated, set $j - 1$ to variable M. ■

In the example of multiple hashing shown in Subsection 3.1, the main processing (entering the keys) is amalgamated to the steps of FOL because of efficiency, but the main processing is not included in FOL1 (the above algorithm), to make the algorithm multi-purpose.

The following condition must hold for the sake of the correctness of FOL1.

■ The exclusive label storing condition (ELS condition)

One of the multiple labels written into one work area is stored correctly. That means, if two labels *la* and *lb* are written into the same area in parallel, the stored value is not an amalgam of *la* and *lb*. Which one of these labels is stored is arbitrary. ■

ELS condition is guaranteed when the label length is equal to or less than the machine word length in normal pipelined vector processors. This condition holds hereafter.*

The lemmas and theorems on FOL1 are shown below.

■ Theorem 1: Termination property

Algorithm FOL1 terminates.

❑ **Proof**: Any label read from a work area is equal to one of the original labels written to the work area by ELS condition. Thus, the read label is equal to at least one of the original labels in Step (2), and S_j is not an empty set for arbitrary j. This means that the number of elements of the index vector V is reduced every time in Step (3). Therefore, V always becomes empty in finite iterations, and FOL1 terminates. ■

The correctness of FOL1 will be proved using two lemmas.

■ Lemma 1: Disjoint decomposition

The disjoint decomposition condition shown in the output specification of FOL1 holds. That means, the union of the output sets, $S_1, S_2, \ldots, S_M$, is equal to the input data set, and $S_i \cap S_j = \varnothing$ for arbitrary i and j, when FOL1 terminates.

❑ **Proof**: If the following conditions hold, the disjoint decomposition condition holds:

(a) each element of the output set S_j ($j = 1, 2, \ldots, M$) is one of the input data,

(b) each input data is an element of an output set, and

(c) the output sets are disjoint.

First, we show condition (a). The index vector V consists of all the pointers, each of which points to each input data on the beginning of the execution. An element, e, of an arbitrary output set S_j is one of the input data, because it is selected from all of the data that the elements of V point to in Step (2).

Next, we will show conditions (b) and (c). V consists of all of and only the pointers or indices to the input data of the beginning of the execution. V becomes an empty set at the termination. The elements are deleted from V in Step (3) immediately after they are added to one of the output sets S_j ($j = 1, 2, \ldots, M$) in Step (2), and all the elements are deleted from V before the execution terminates. That means that all of the data pointed to from V are added to one of the output sets. In addition, the pointer or index to an element of S_j is deleted immediately after the content of S_j is computed, so this pointer or index never belongs to S_m ($m > j$). Thus, the output sets are disjoint. ■

The cardinalities (the numbers of elements) of $S_1, S_2, \ldots, S_M$ are noted by $|S_1|, |S_2|, \ldots, |S_M|$, respectively. Then, the following lemma and theorem hold.

■ Lemma 2

If d_k and d_l ($k \neq l$) are arbitrary elements of an output set S_k, then d_k and d_l are in different areas (d_k and d_l are different data).

❑ **Proof**: The lemma is proved by contradiction. If d_k and d_l are in the same area, the pointers or indices to them, which are the elements of V, have the same value. The labels assigned to these elements are stored

---

* There is a type of algorithms, which FOL should be applied to, in which the order of multiple processings for *one* data must be preserved. It is possible to modify FOL1 so as to eliminate the above process condition and to construct vectorized algorithms of such a type. That is, if processings P_i are applied to data d_i ($= v_i \rightarrow d$) and processing P_j are applied to data d_j ($= v_j \rightarrow d$) where d_i and d_j are equivalent (i.e., in the same storage area), and the execution of P_i precedes that of P_j in the sequential execution, then ELS condition can be replaced by a stronger condition so that the relation $k > l$ holds for the output sets S_k and S_l, where $d_i \in S_k$ and $d_j \in S_l$.

For the vector processor S-820, higher-performance list-vector store instruction (VIST (Vector Indirect STore) instruction), which satisfies ELS condition, can be used for writing labels in FOL1. However, VSTX (Vector STore indeXed) instruction, which is slower but guarantees the order of storing the vector elements (a stronger condition than ELS condition), can be used instead of VIST to satisfy the above relation, $k > l$, and to eliminate the above condition.

into the same work area. One of these labels is overwritten by the other, so d_k and d_l are included in different output sets in Step (2). This is a contradiction, so the lemma is concluded. ■

■ Theorem 2: Correctness

The output conditions hold when FOL1 terminates.

❑ **Proof**: This theorem is proved by Lemmas 1 and 2. (Lemma 2 guarantees that the output sets are parallel-processable.) ■

One more theorem is given but the proof is omitted.

■ Theorem 3

The following relation always holds: $|S_1| \geq |S_2| \geq \ldots \geq |S_M|$, and $M = 1$ (the number of iterations in FOL1 is equal to one) when the input data does not have duplications. ■

The allocation of the work area used in FOL1 is explained. Normally, the work area can share storage with the area used by the main processing, because it does not matter whether the value held in the area pointed to by the elements of V is destroyed before FOL1 is applied by writing labels, and because there is no possibility that the wrong value, which is not a correct label, is read in the process of overwriting detection. It does not matter whether the value is destroyed because the main processing will rewrite the storage area where the labels are written by FOL1. Conversely, a sufficient condition that the main processing always rewrites the work area, where the labels are written, or a weaker condition must hold. There is no possibility that the wrong value is read, because there is no possibility that, while reading labels in Step (2), the labels are read from an area where no labels were written, because ELS condition holds and the labels are read through the same pointers or indices used when writing the labels.

Because the size of each work area is $\log_2 N$ bits or more, the shared area must be extended when the main processing requires less area. The size must be $\log_2 N$ bits or more because the work area must have enough capacity to hold one of N different labels.

The performance of FOL1 is examined next. The sequentially processed part of the main processing is not accelerated by FOL. On the contrary, the execution of this part becomes slower because of the overhead of the detection of parallel-processable data. Consequently, the sequential execution is better than FOL in a processing where most of the data cannot be processed in parallel. However, if the sharing rarely occurs and most of the data can be processed in parallel, FOL is promising.

The following theorem guarantees that the execution time of FOL1 is $O(N)$ when the number of sharing is small.

■ Theorem 4: $O(N)$ execution time

If condition $|S_1| \gg \sum_{i=2}^{M} |S_i|$ holds, then the execution time of FOL1 is $O(N)$.

❑ **Proof**: If the above condition holds, the execution time of Step (4), which depends on $|S_i|$ (i = 2, 3, …, M), can be ignored compared with the sum of the execution time of Steps (1) and (2) which depend on $|S_1|$. Then the execution time of FOL1 is $O(N)$, because the execution time of both Steps (1) and (2) is $O(N)$. ■

In particular, the execution time of FOL1 is $O(N)$ when there are no duplications in the input data, by Theorems 3 and 4.

A lemma and two theorems are given. Theorem 5 guarantees the best performance in a sense, when condition $|S_1| \gg \sum_{i=2}^{M} |S_i|$ does not hold.

■ Lemma 3

If there are M' duplications in the input data, all of which are the same, including the original data (i.e., there is a storage area which is shared by M' input data), and there are no more than M' duplications, the number of output sets, M, is equal to M'.

❑ **Proof**: If there are M' duplications, V has M' elements, which point to the same storage area containing the duplicated data, at the beginning of FOL1. As explained in the proof of Theorem 1, there always exists a label that coincides with the original in Step (2). In addition, each input data has a unique label and only one of the labels is read repeatedly M' times, so, in the labels of the M' elements of V, there is exactly one label that coincides. Therefore, exactly one of these elements is deleted from V every time Step (3) is executed. The number of iterations in FOL1 is equal to M. Thus, M' is equal to M. ■

■ Theorem 5: Minimum decomposition

If $T_1 \cup T_2 \cup \ldots \cup T_{M''}$ is an arbitrary decomposition of the input data, i.e., $\bigcup_{i=1}^{M''} T_i = \{d_1, d_2, \ldots, d_N\}$, where the elements of T_i is parallel-processable for arbitrary i, and the number of output sets of FOL1 is M, then $M'' \geq M$. That means that the number of output sets of FOL1 is minimum.

❑ **Proof**: The duplicated data does not belong to the same output set because, otherwise, the output set is not parallel-processable. Thus, if there are M' duplications in the input data, the number of parallel-processable sets is no less than M' for arbitrary decomposition. The number of output sets of FOL1 is also M' by Lemma 3. Thus, the number of output sets of FOL1 is minimum. ■

■ Theorem 6: Worst execution time

The execution time of FOL1 is $O(N^2)$ when the following condition holds: $|S_1| = |S_2| = \ldots = |S_M| = 1$.

❑ **Proof**: The execution time of the single iteration in FOL1 is mainly dependent on that of Steps (1) and (2), and they are in proportion to the number of the elements of V. The number of elements of V decreases one by one under the above condition. So the execution time of the j-th iteration is $(N - j + 1)\,t + c$, if the sum of the execution time of Steps (1) and (2) is t and c is a constant, and the total execution time is approximately $\sum_{j=1}^{N} (N - j + 1)\,t + C$, which means it is $O(N^2)$. ■

The application area of FOL1 is considered next. FOL1 is a vectorization/parallelization method which can be used in a wide range of applications. Multiple hashing is a typical multiple data processing where a small number of sharings exist, but the same condition holds in many applications, for example, address calculation sorting and parallel rewriting of lists, trees (DAGs) or graphs with sharing, as illustrated in Figure 3.

Finally, a method of improving the execution time of FOL1 by simplifying its process is examined. The following simplified method can be applied when there is no duplication in the values to be written into the area pointed to by the elements of the index vector V. In this case, these values can be used for labels, and the label writing and the main processing (i.e., writing the values) can be performed at the same time. For multiple hashing, the above condition holds when the keys are unique and used as labels.

3.3 FOL for rewriting multiple data per unit process

FOL1 can be applied when only one data is rewritten in a unit process. An FOL algorithm, which is applied when multiple data are rewritten in a unit process, is as follows.

■ Algorithm: The Overwritten-Label Filtering Method 2 (FOL*)

❑ Input

This algorithm inputs index vectors $V_1, V_2, \ldots, V_L$. The elements of these vectors are pointers or indices to storage areas containing data $d_{i1}, d_{i2}, \ldots, d_{iL}$ ($i = 1, 2, \ldots, N$), respectively, where there may be duplicated data. The storage area pointed to by v, an element of V_k, is noted by $v\rightarrow$, and the data stored in $v\rightarrow$ is noted by $v\rightarrow d$.

❑ Output

This algorithm outputs sets of tuples $\langle d_{i1}, d_{i2}, \ldots, d_{iL} \rangle$ ($i = 1, 2, \ldots, N$) of parallel-processable data: $S_1, S_2, \ldots, S_M$ (the value of M is obtained as an execution result of this algorithm), where $\bigcup_{j=1}^{M} S_j = \{\langle d_{i1}, d_{i2}, \ldots, d_{iL} \rangle \mid i = 1, 2, \ldots, N\}$ and $S_1, S_2, \ldots, S_M$ are disjoint sets, i.e., $S_i \cap S_j = \varnothing$ (***disjoint decomposition condition***).

❑ Process conditions

- Processing P_i ($i = 1, 2, \ldots, N$) is applied to a data tuple $\langle d_{i1}, d_{i2}, \ldots, d_{iL} \rangle$ by vector operation after the application of this algorithm. All the data included in an output set S_j ($j = 1, 2, \ldots, M$) may be processed in parallel or in any order, but any two data belong to different output sets may not be executed in parallel. The execution order must not affect the correctness of the result.
- If va is an arbitrary element of V_f and vb is an arbitrary element of V_g before the execution where $f, g \in \{1, 2, \ldots, L\}$, $va\rightarrow$ and $vb\rightarrow$ are possibly the same storage area. That means $V_1, V_2, \ldots, V_L$ may have the same pointers as their elements.
- A work area to be used in this algorithm is reserved for each storage area pointed to by each element of $V_1, V_2, \ldots, V_L$. The work area pointed to by v is noted by $v\rightarrow w$.

❑ Procedure

(0) Preprocessing: Set 1 to variable j. Assign a unique label to each element of index vectors V_k ($k = 1, 2, \ldots, L$). That means, if la is the label of an arbitrary element of V_{k1} and lb is the label of an arbitrary elements of V_{k2} and these elements are different, condition $la \neq lb$ must hold. The labels may be assigned before the execution time.

(1) Writing labels: $v_{k1}, v_{k2}, \ldots, v_{kn}$ are assumed to be the elements of the index vector V_k, where n is the number of elements of $V_1, V_2, \ldots, V_L$ (where all of them have the same number of elements). For $k = 1, 2, \ldots, L$, perform the following process. Write the labels of $v_{k1}, v_{k2}, \ldots, v_{kn}$ into the work areas $v_{k1}\rightarrow w, v_{k2}\rightarrow w, \ldots, v_{kn}\rightarrow w$, respectively. The execution order is arbitrary, and the labels may be processed in parallel.*

(2) Detection of overwriting: For $k = 1, 2, \ldots, L$, perform the following process. Read the labels from the work areas $v_{k1}\rightarrow w, v_{k2}\rightarrow w, \ldots, v_{kn}\rightarrow w$ and compare them with the labels of $v_{k1}, v_{k2}, \ldots, v_{kn}$,

* A deadlock may occur in Step (4). A solution to this problem will be explained later.

respectively. Step (1) must be completed before label reading. If $v_i \rightarrow w$ is not equal to the label of v_i, it means that v_i is overwritten. The set of all the data tuples $\langle d_{i1}, d_{i2}, \ldots, d_{iL} \rangle$ whose elements are the data pointed to by the i-th elements of $V_1, V_2, \ldots, V_L$, respectively, except the tuples which contain a data, for which the equality check has failed, is the parallel-processable set S_j. More exactly, set S_j is defined as follows: $l_{1j}, l_{2j}, \ldots, l_{Lj}$ are assumed to be the labels assigned to the index vector elements v_{1i}, $v_{2i}, \ldots, v_{Li}$, then S_j is the set of all the tuples $\langle v_{1i} \rightarrow d, v_{2i} \rightarrow d, \ldots, v_{Li} \rightarrow d \rangle$ $(i = i_1, i_2, \ldots, i_n)$ where condition $v_{1i} \rightarrow w = l_{1i} \wedge v_{2i} \rightarrow w = l_{2i} \wedge \ldots \wedge v_{Li} \rightarrow w = l_{Li}$ holds.

(3) Updating control variables: Add 1 to j. For $k = 1, 2, \ldots, L$, delete the pointers or indices pointing to data in a tuple in S_j, from V_k. The number of elements of V_k is reduced for each k.

(4) Repetition: Repeat the above Steps (1) to (3) until V_k becomes empty. (Testing one of $V_1, V_2, \ldots, V_L$ is enough because all of them have the same number of elements.) When terminated, set $j - 1$ to variable M. ■

ELS condition must hold in Step (1) as in FOL1. In addition, a deadlock may occur unless another appropriate condition is added on the label writing order, as explained in Step (1). That means, in the case that an appropriate writing order of the labels cannot be guaranteed, the relation in Step (2) does not hold for any element of the index vectors. Then S_j may be an empty set and the control does not exit from the loop in FOL*.

A method to avoid the above problem is explained. The elements of the index vectors except the last ones are written by vector instructions in parallel, but the last elements are written by scalar instructions sequentially after the execution of the vector instructions. It is asserted that there are no shared elements among the last elements of the index vectors. Then the relation shown in Step (2) holds for the last elements, and S_j is prevented from being empty. However, the above method may cause a significant decrease of parallelism, and the acceleration ratio may become less than 1.0. So, a better method should be developed.

The proofs of the termination property and the disjoint decomposition condition are omitted because they can be proved in the same way as the case of rewriting single data shown in Subsection 3.2.

The performance of FOL* is examined next. When the number of data rewritten in a unit process, i.e., L, is large, the execution time of FOL* becomes larger compared with the main processing, and the acceleration ratio* of the total process is low. Thus, FOL* is considered to be practical only when L is less than five or so. The value of L is two in the case of the operation tree rewriting shown in Section 2, so the vectorized algorithm will be practical in this case.

4. Applications of FOL

Three applications of FOL1 and the resulting performances are shown in this section.

4.1 Multiple hashing

There are two ways of collision resolution in hashing: *open addressing* and *chaining* [Knu 73]. Figures 4 and 7 apply chaining. The algorithm for multiple hashing using open addressing in Kanada [Kan 90] is based on a specialized version of FOL, which is called the "overwrite-and-check method." The FOL algorithms shown in Section 3 are abstractions of this algorithm.

The result of an optimized version of Kanada's multiple hashing algorithm is shown in this paper. The method of subscript recalculation in the case of collisions in this algorithm is different from the original [Kan 90]. In the original algorithm, the n-element vector of subscripts, $hashedValue[1:n]$, is computed from the old subscripts, which are initially equal to the hashed values, as follows:

```
$hashedValue[1:n]$ :=
    ($hashedValue[1:n]$ + 1) **mod** $size(table)$;
    — All of the elements are incremented by one.
```

However, if three or more keys in the key vector collide, the keys failed to be entered cause collisions again when tried to be entered again, because the subscripts are the same again. Thus the optimized algorithm recalculates the vector of subscripts as follows:

```
$hashedValue[1:n]$ :=
    ($hashedValue[1:n]$ + ($key[1:n]$ & 31) + 1) **mod**
    $size(table)$;
    — "&" means bitwise "and" operation.
    — $size(table) > 32$ is asserted.
```

The optimized algorithm is coded in Fortran** and executed on the Hitachi S-810 [Nag 84], which is an older-generation vector processor than the S-820. All of the innermost loops are vectorized. **Figures 8** and **9** display the CPU time and acceleration ratio of multiple hashing where the table sizes are 521 and 4099. The horizontal axes show the load factor (the ratio of the filled table entries) after entering the keys. The acceleration ratio reaches the maximum value, 5.2 or

* The acceleration ratio means the ratio of the vectorized total execution time and the *original* sequential execution time.

** However, because the vectorized algorithm cannot be written in standard Fortran, the program violates its semantics using vectorization forcing option (`*VOPTION`).

12.3, when the load factor is 0.5. The reason that the acceleration ratio increases when the load factor is less than 0.5 is that the vector length is proportional to the load factor. The reason that the acceleration ratio decreases when the load factor is between 0.5 and 1.0 is that the effect of reducing the performance caused by the increase of sequentiality is larger than the acceleration effect caused by the increase of the vector length. However, the acceleration ratio converges to a value larger than 1.0 when the load factor gets closer to 1.0. This fact means that the vector processing is faster than the sequential processing, because the parallelism still remains when the load factor gets closer to 1.0.

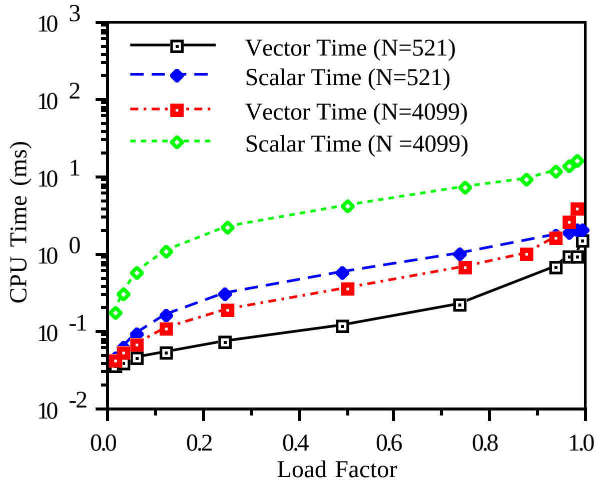


Figure 8. CPU time of multiple hashing into an empty hash table by S-810 (*N* : table size)

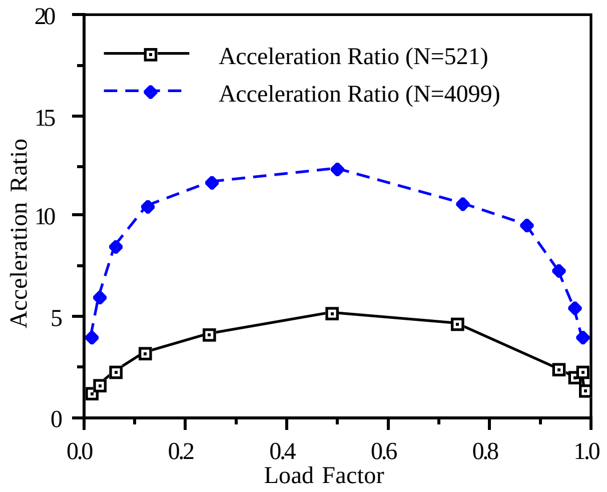


Figure 9. Acceleration ratio of multiple hashing into an empty hash table by S-810 (*N* : table size)

The results of the optimized algorithm shown in Figures 8 and 9 are better than those of the original [Kan 90] in that the acceleration ratio is larger when the load factor is between 0.5 to 0.98. This is the result of improving the method of subscript recalculation for colliding keys.

4.2 Address calculation sorting and distribution counting sort

An FOL1-based algorithm of an address calculation sorting [Flo 60, Gon 84] and the results of the address calculation sorting and a distribution counting sort [Knu 73] are shown in Kanada [Kan 90]. The summary of the results (the same results as shown in [Kan 90]) is shown in **Table 1**. The maximum acceleration ratio of the address calculation sorting is more than ten.

Table 1. CPU time and the acceleration ratio of *O*(*N*) sorting algorithms

Algorithm	*N*	S-810/20 CPU time (μs)		Acceleration ratio
		Sequential	Vectorized	
Address Calculation Sorting*	2^6	289	110	2.62
	2^{10}	4,286	560	7.65
	2^{14}	66,955	5,215	12.84
Distribution Counting Sort**	2^6	12,206	1,522	8.02
	2^{10}	13,072	1,738	7.52
	2^{14}	30,089	5,667	5.31

* The size of the work array *C* is 3*n*.
** The size of the work array is 2^{16}, which is the range of the data.

4.3 Entering multiple data into a binary tree

An algorithm that enters multiple data into a *linked* binary tree using FOL1 has been developed. The tree is not balanced in this algorithm. **Figure 10** displays a preliminary result of a performance evaluation on the S-810. The horizontal axis indicates the number of uniformly random keys entered into the tree, and the vertical axis indicates the acceleration ratio. The tree has *Ni* elements, each of which contains a random key, before entering the keys. The reason why an empty tree is not used for the benchmarking is that it is too disadvantageous for vector processing because all the keys to be entered conflict when the tree is empty. The number of trials for each plotted point is only one, so this result is not very reliable. However, we can conclude that the average acceleration ratio is more than 1.0, though it is not a factor of ten, if the initial tree size is not very small and the amount of data to be entered is not very small.

5. Related Works

Appel and Bendiksen's vectorized garbage collection algorithm [App 89] implicitly includes a very specialized version of FOL. Miki et. al. use a similar technique in the vectorized LSI routing algorithm [Mik 91]. In both algorithms, the first output set S_1 is implicitly computed. The output sets S_2, S_3, …, S_M are not computed because they are unnecessary in these algorithms.

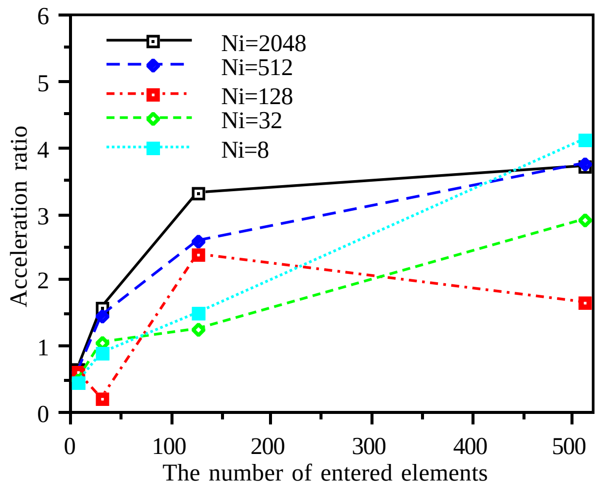


Figure 10. Acceleration ratio of entering multiple data into a binary tree by S-810

6. Conclusion

The *filtering-overwritten-label method* (FOL) given in this paper enables vector processing over a wide range of applications which include rewriting data with shared elements or possibly rewriting the same data element two or more times. The evaluation results show that the application of FOL to multiple hashing and to an address calculation sorting accelerates the execution by a factor of ten. FOL is a promising vector-processing technique for processing lists, trees, graphs or other symbolic processings on pipelined vector processors or SIMD parallel processors such as CM-2 (Connection Machine).

The main focus of future works will be to apply FOL to various symbolic algorithms including tree re-balancing and graph rewriting.

Acknowledgements

The author wishes to thank Dr. Sakae Takahashi and Seiichi Yoshizumi of Hitachi Ltd. for their continuing support of his research, Shun'ichi Torii of Hitachi Ltd. for his comments that led to significant improvements in this paper, and Dr. Michiaki Yasumura of Keio University, Keiji Kojima and Masahiro Sugaya of Hitachi Ltd. for their suggestive comments.

References

[App 89] Appel, A. W., and Bendiksen, A.: Vectorized Garbage Collection, *J. Supercomputing*, Vol. 3, pp. 151–160 (1989).

[Flo 60] Flores, I.: Computer Times for Address Calculation Sorting, *J. ACM*, Vol. 7, No. 4, pp. 389–409 (1960).

[Gon 84] Gonnet, G. H.: *Handbook of Algorithms and Data Structures*, Addison-Wesley (1984).

[Kam 83] Kamiya, S., Isobe, F., Takashima, H., and Takiuchi, M.: Practical Vectorization Techniques for the "FACOM VP," *Information Processing '83*, pp. 389–394 (1983).

[Kan 88] Kanada, Y., Kojima, K., and Sugaya, M.: Vectorization Techniques for Prolog, *1988 ACM Int. Conf. on Supercomputing*, pp. 539–549 (1988).

[Kan 89a] Kanada, Y., and Sugaya, M.: Vectorization Techniques for Prolog without Explosion, *Int. Joint Conf. on Artificial Intelligence '89*, pp. 151–156 (1989).

[Kan 89b] Kanada, Y., and Sugaya, M.: A Vector Processing Method for List Based on a Program Transformation, and its Application to the Eight-Queens Problem, *Transactions of Information Processing*, Vol. 30, No. 7, pp. 856–868 (1989), in Japanese.

[Kan 90] Kanada, Y.: A Vectorization Technique of Hashing and its Application to Several Sorting Algorithms, *PARBASE-90*, pp. 147–151, IEEE (1990).

[Koj 87] Kojima, K., Torii, S., and Yoshizumi, S.: IDP — A Main Storage Based Vector Database Processor, *1987 Int. Workshop on Database Machines*, pp. 60–73 (1987).

[Knu 73] Knuth, D. E.: Sorting and Searching, *The Art of Computer Programming*, Vol. 3, Addison-Wesley (1973).

[Mik 91] Suzuki, K., Miki, Y., and Takamine, Y.: *An Acceleration of Maze Algorithm Using Vector Processor*, Technical Report CAS 91-17, Vol. 91, No. 55, pp. 23–28, Institute of Electronics, Information and Communication Engineers (1991), in Japanese.

[Nag 84] Nagashima, S., et al.: Design Consideration for High-Speed Vector Processor: S-810, *IEEE Int. Conf. on Computer Design*, pp. 238–242 (1984).

[Tor 88] Torii, S., Kojima, K., Kanada Y., Sakata, A., Yoshizumi, S., Takahashi, M.: Accelerating Non-Numerical Processing by An Extended Vector Processor, *4th Int. Conf. on Data Engineering*, pp. 194–201 (1988).