Using Tries to Eliminate Pattern Collisions in Perfect Hashing
Marshall D. Brain and Alan L. Tharp

Abstract—Many current perfect hashing algorithms suffer from the problem of pattern collisions. In this paper, a perfect hashing technique that uses array-based tries and a simple sparse matrix packing algorithm is introduced. This technique eliminates all pattern collisions, and because of this it can be used to form ordered minimal perfect hash functions on extremely large word lists.

This algorithm is superior to other known perfect hashing functions for large word lists in terms of function building efficiency, pattern collision avoidance, and retrieval function complexity. It has been successfully used to form an ordered minimal perfect hashing function for the entire 24,481 element Unix word list without resorting to segmentation. The item lists addressed by the perfect hashing function formed can be ordered in any manner, including alphabetically, to easily allow other forms of access to the same list.

Index Terms—Perfect hashing, minimal perfect hashing, hashing tries, sparse array packing.

I. INTRODUCTION

HASHING is a technique used to build modifiable data structures that maintain efficient retrieval times for all items in the structure. As items are added to the structure, the attempt is made to maintain $O(1)$ access time at retrieval. This efficiency is achieved by placing each unique item that is added at a unique address in the structure. A number of algorithms have been suggested, but no current hashing algorithm can maintain $O(1)$ retrieval times over the lifetime of the data structure: Eventually multiple items collect at identical addresses, and these collections must be handled in some way. All current algorithms used to resolve these collections have a retrieval penalty, which causes the retrieval efficiency to rise above the ideal $O(1)$ efficiency.

A number of perfect hashing algorithms have been created recently that do allow the $O(1)$ retrieval of all items in the data structure as long as the items are static. These algorithms all start with an original and static list of items and use various techniques to build a hash function that is specific to that set of items. A perfect hashing function may perform multiple computations or table look-ups to produce the unique retrieval address, but each item in the list can be retrieved with $O(1)$ efficiency.

Perfect hashing functions were used as early as 1977 to allow $O(1)$ retrieval of items such as month names [23]. Algorithms such as Cichelli's algorithm [6], which is capable of handling words lists of up to about 50 words, have been used for tasks such as the $O(1)$ retrieval of Pascal reserved words within a Pascal compiler.

Interest in perfect hashing algorithms has risen recently with the advent of extremely large static databases, particularly those held on CD-ROM's. CD-ROM's are a nonwriteable medium, which guarantees static data. CD-ROM's also have slow track seek times, which makes multiple accesses to the disk extremely frustrating for the user of the CD-ROM. Perfect hashing functions have been used in this environment to provide quick access to large medical dictionaries [9] and other word lists. The perfect hashing function allows any word to be accessed from the disk with only one disk access operation. Unfortunately, many current perfect hashing algorithms are severely limited in the number of words that can be processed.

Future application of perfect hashing functions to large static databases will continue. On-line periodical indexes and card catalogs, especially those committed to CD-ROM, will continue to demand the high-efficiency retrieval capabilities offered by perfect hashing. Perfect hashing algorithms must expand to handle immense item lists of this type.

A new perfect hashing technique based on compressed tries has been developed. It can be used to form ordered minimal perfect hashing functions for extremely large word lists. The new technique is unique in its speed and efficiency, as well as in its immunity to pattern collisions.

II. AVAILABLE PERFECT HASHING ALGORITHMS

Sprugnoli [23] introduced the first perfect hashing algorithm. Two techniques—the quotient reduction method and the remainder reduction method—were developed to allow perfect hashing of relatively small sets of words (10–12 words). These methods both rely on the algorithm's ability to create a simple formula of the form $H = (W - C1)/C2$ (quotient reduction method) or $H = (FLOOR((C1 + W \times C2 mod C3)/C4))$ (remainder reduction method), where $H$ is the hash value, $W$ is the word to be hashed, and $C1, C2, C3, and C4$ are constants calculated by the algorithm, for the word set in question. For larger sets of words, Sprugnoli suggests a segmentation process to break the list into smaller subsets. Sprugnoli's methods are unique in the fact that they do not require auxiliary tables of information at retrieval time—only the four constants must be stored.

Jaeschke [14] created a technique called reciprocal hashing. This technique is similar to Sprugnoli's in that the perfect hashing functions are relatively simple, but like Sprugnoli's...
algorithm, Jaeschke’s algorithm cannot handle more than about 20 words without segmentation.

Cichelli [6] developed an extremely straightforward minimal perfect hashing algorithm that can be used on word lists up to about 50 words in size. Cichelli’s method makes use of a table of 26 integers that is used at retrieval time. The perfect hashing algorithm uses a brute-force technique to find a set of values for the table that allows each word to be hashed to a unique address. For larger word sets Cichelli suggests segmentation. Cichelli’s algorithm is extremely sensitive to pattern collisions. We introduce the term pattern collisions to describe the situation that occurs when two words cannot be handled simultaneously by a perfect hashing function. If the technique used by the perfect hashing function to determine the address for a word causes two words to be indistinguishable, a pattern collision has occurred. For example, a perfect hashing function formed by Cichelli’s algorithm uses the first and last characters of the word being hashed, along with the length of the word, to decide on the hash address. Because of this, any two words with the same first and last character and length cannot be distinguished, and therefore cannot be handled simultaneously by Cichelli’s algorithm (e.g., “dig” and “dug”).

Karplus and Haggard [16] generalized Cichelli’s algorithm using graph theory methods to hash word sets of up to almost 700 words. Karplus also notes experimental results that seem to indicate an extremely good efficiency for the algorithm (see Table 1). However, the algorithm has not been successfully applied to word lists greater than 700 words, and the words in the sample lists [15] seem to be carefully chosen. For example, one list contains the words “Tannarive,” “thuggee” and “thyrogblobulin,” but omits more common words such as “the,” “there,” and “through.”

Chang [7] built an ordered minimal perfect hashing algorithm using the Chinese remainder theorem. Ordered functions allow items in the static list to be arranged in any order (e.g., alphabetically) as the perfect hashing function is built. Chang’s algorithm is unique in this respect.

Du, Hsieh, Jea, and Sheith [8] created a perfect hashing function based on segmentation and random hashing functions. Word lists of up to 300 words were successfully hashed, though no execution times are revealed.

Cercone, Krause, and Boates [5] started with Cichelli’s algorithm and created three different perfect hashing algorithms. One forms segments based on word length and then applies Cichelli’s algorithm to each segment. The results for the subsets are then combined into a single table. The remaining two algorithms are more esoteric. The best of the three algorithms is able to handle up to 500 words, but the results are not minimal (one-third of the final table is unused).

Sager [22], also trying to improve on Cichelli’s algorithm, used a graph theoretic approach to create minimal perfect hashing functions of sets of words as large as 512. Sager’s work was later improved by Fox, Chen, Heath, and Daoud [9] to form minimal perfect hashing functions of lists containing up to 1000 words. Both of these algorithms expand on Cichelli’s table size considerably to handle large word sets. To increase the number of words hashed, Fox suggests segmentation of the word list.

Larson and Ramakrishna [21], [19] created a perfect hashing algorithm for external files. The technique uses a segmentation strategy to divide a large file into smaller groups. Each group is then perfectly hashed using standard techniques.

Fox, Chen, Heath, and Daoud [10] have demonstrated a perfect hashing algorithm capable of handling large word lists. The algorithm is $O(n)$ at build time and uses very little space to store the perfect hashing tables needed at retrieval time. However, this algorithm does not produce an ordered perfect hashing function, and it is also fairly expensive at retrieval time. See subsequent sections for discussions of these issues.

Further discussion of perfect hashing functions can be found in [25] and [20]. Berman [2] presents theoretical analysis on the space tradeoffs involved in creating perfect hashing functions.

### III. Perfect Hashing Issues

Perfect hashing algorithms are used to build perfect hashing functions. The perfect hashing function normally consists of a set of values (often stored in a table or array) that are calculated by the perfect hashing algorithm. The formation of the perfect hashing function by the perfect hashing algorithm occurs once for a given, static item list. The perfect hashing function is then used for each retrieval of all items from the item list.

Several issues have become prominent in perfect hashing research. These issues include the following.

1) The ability of the perfect hashing algorithm to form minimal perfect hashing functions.
2) The ability of the algorithm to form ordered perfect hashing functions.
3) The memory requirements of the perfect hashing function at retrieval time.
4) The amount of CPU time required by the perfect hashing algorithm while the perfect hashing function is being built.
5) The amount of CPU time required by the perfect hashing function for each retrieval.
6) The sensitivity of the perfect hashing algorithm and function to pattern collisions.

A hash function’s ability to form “minimal” item lists is extremely important because of space considerations. In a minimal list, the items addressable by the perfect hashing function are arranged contiguously. Minimal lists save considerable storage space if the items addressed by the perfect hashing function are large records of information. Most of the algorithms discussed previously produce minimal functions,

### Table 1: Build Order of Different Algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Reference</th>
<th>Build</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cichelli</td>
<td>[6]</td>
<td>$O(c^r)$</td>
</tr>
<tr>
<td>Karplus</td>
<td>[15]</td>
<td>$O(r^{c^2})$*</td>
</tr>
<tr>
<td>Sager</td>
<td>[22]</td>
<td>$O(r^2)$</td>
</tr>
<tr>
<td>Fox</td>
<td>[9]</td>
<td>$O(r^3)$</td>
</tr>
</tbody>
</table>

*Derived from experimental results rather than theoretical analysis.
BRAIN AND THARP: PATTERN COLLISIONS IN PERFECT HASHING

but most also allow nonminimal functions to be formed as well. In general, less time is required to build nonminimal perfect hashing functions.

The order of the item list is sometimes important. For example, the item list must be accessible by the perfect hashing function but may also need to be accessible alphabetically or sequentially. Perfect hashing algorithms that allow arbitrary and specific orderings (e.g., alphabetical) of the item list once the perfect hashing function is formed are called ordered functions. Chang's algorithm [7] creates an ordered perfect hashing function, while most others do not.

Another issue important to perfect hashing functions is the amount of memory required by the function at retrieval time. All perfect hashing functions require the use of a precalculated set of values to handle retrieval. The values are determined when the perfect hashing function is built. For example, Sprugnoli [23] calculates four constants \( C_1, C_2, C_3, \) and \( C_4 \) that are used at retrieval. Cicelli [6] calculates a table of 26 values that are accessed for retrieval. In Cicelli's algorithm, the first letter and last letter of the word to be retrieved are used as indexes into the table. The two values found in the table are summed together with the length of the word to form the hash address. The purpose of Cicelli's algorithm is to find values for the table that cause no address collisions for all of the words in the static item list. The amount of space required for the table tends to grow with the number of items hashed. For example, Sager's algorithm [22] requires 4 bytes of storage space in the table for each item in the static list of items. This table is built by the perfect hashing algorithm once, and then stored in memory and used for each retrieval performed by the perfect hashing function.

Each perfect hashing algorithm has two completely different time efficiency aspects that must be monitored. One of these aspects is the retrieval efficiency. Because a perfect hashing function maps a key to a unique address, it has \( O(1) \) retrieval efficiency by definition. The data are stored on auxiliary memory, but the hashing function is processed in primary memory. Because the access time for auxiliary memory is several orders of magnitude greater than the access time for primary memory or the execution time for an instruction, the operations performed by the perfect hashing function are insignificant compared with an auxiliary memory access. In a perfect hashing function, each item in the list can be retrieved with \( O(1) \) accesses of auxiliary memory.

The second time efficiency aspect involves actually building the perfect hashing function. For any algorithm, this is the one-time cost of constructing a perfect hashing function for a given set of items. For example, Cicelli's algorithm has an \( O(c^r) \) build time (where \( c \) is some constant and \( r \) is the the number of words to be hashed), while Sager's minicycle algorithm has an \( O(r^5) \) build time (where \( r \) is the number of words hashed). Table I shows the build order of some of the algorithms mentioned.

Many of the algorithms described above are sensitive to pattern collisions to a certain degree. As discussed in an earlier example (and in [3]), any two words with the same first and last character and length (e.g., "dig" and "dug") cannot be handled simultaneously by Cicelli's algorithm. Pattern collisions are normally handled by segmenting the original word list into a set of sublists, using a hashing function to divide the words between the different segments. The hashing function used to form the segments is chosen so that words with pattern collisions are placed in different segments. This segmentation is accompanied by a performance penalty.

Finally, the complexity of the functions used for retrieval is important. Although all perfect hashing functions have an \( O(1) \) (constant) retrieval time, the amount of CPU time required for each retrieval should be minimized. Equations (1)-(6) show some of the different functions used for retrieval in perfect hashing functions. Cicelli's algorithm has an extremely simple and efficient retrieval function, as shown in (2). Sager uses somewhat more complicated functions, as shown in (3)-(6). (Note that this set of equations comes from [22] and is required for large word lists. For smaller lists, simpler equation sets are possible. The listed equation set, or one of similar complexity, is necessary to eliminate pattern collisions in large word sets because many characters must be examined to avoid the collisions.) As pointed out by Sager [22], Chang [7] uses retrieval functions that are extremely time consuming, making them impractical.

**Sprugnoli:**

\[
H = \left\lfloor \left( C_1 + W \ast C_2 \right) \mod C_4 \right\rfloor / C_4 \tag{1}
\]

**Cicelli:**

\[
\]

**Sager:**

\[
\begin{align*}
H_0 &= \left( L E N G T H ( W ) + \left( S U M ( O R D ( W [ I ] ) ) \right) \right), I \\
H_1 &= \left( S U M ( O R D ( W [ I ] ) ) \right), I \\
H_2 &= \left( S U M ( O R D ( W [ I ] ) ) \right), I \\
H &= H_0 + G[ H_1 ] + G[ H_2 ] \mod N
\end{align*} \tag{3-6}
\]

In the equations shown above, \( W \) represents the word being retrieved. \( L E N G T H ( W ) \) is the number of characters in the word. \( W[x] \) is the \( x \)th character of the word. \( G \) represents a table of values calculated by the given perfect hashing algorithm, indexed using the value shown. \( C_1, C_2, C_3, C_4, \) and \( R \) are constants. \( H \) represents the hash address calculated by the perfect hashing function.

**IV. A SIMPLE PERFECT HASHING FUNCTION**

A typical application of a perfect hashing function is the \( O(1) \) retrieval of Pascal reserved words by a Pascal compiler. The Pascal reserved words are static and are known in advance, and this knowledge is used to build a specific perfect hashing function for the reserved word set. When a word is to be retrieved, the perfect hashing function is applied to the word. The use of perfect hashing functions can make token recognition for a compiler efficient.
build perfect hashing functions for this application. These functions cover a wide range of complexities, though the simpler algorithms often have significant disadvantages. In this section a simple algorithm will be described and used to demonstrate perfect hashing, and also to demonstrate the six issues described above.

An extremely simple perfect hashing function for Pascal reserved words could be implemented using a 2-D array of integers, as shown in Fig. 1. The first and last characters are used as indexes into the 2-D array, and the integer at the indexed location points to the word in the word list (normally the word list would contain information needed by the compiler to process/compile the given reserved word).

In Fig. 1, a list of seven Pascal reserved words has been hashed. To look up the word AND using this arrangement, the first letter, A, is used as the row index into the array, while the last letter, D, is used as the column index. At the location addressed by A and D is the value 1, which is used to access the record referenced by the word AND in a word list. As can be seen, the retrieval time for any of the words in the word list is constant, regardless of the number of words in the array, giving this perfect hashing function an $O(1)$ retrieval time. When using this algorithm on a very large word list, the array would be stored in primary memory and the word records would be stored on disk, allowing the word records to be accessed with a single disk probe.

The job of the perfect hashing algorithm is to build the 2-D array (a one-time operation). The perfect hashing function (used many times) would use this 2-D array along with the function shown in (7).

$$\text{Hash address} = \text{array[first letter, last letter]}$$ (7)

In (7), first letter and last letter are the first and last letter of the word being hashed. This is an extremely efficient retrieval function compared with other available algorithms.

This perfect hashing algorithm forms ordered minimal perfect hashing functions. The word list contains no gaps between the words in the list—they are all contiguous. The word list shown in Fig. 1 can also be ordered in any way.

This particular perfect hashing algorithm would also have an $O(n)$ build time. When the perfect hashing function is being built, a single value is placed in the 2-D array for each word in the word list.

The memory usage for this perfect hashing function is very inefficient because many entries in the 2-D array are empty. In the example shown in Fig. 1, 669 of the 676 entries in the 2-D array are unused (about 1% memory efficiency). For the seven words handled by this perfect hashing function, 194 bytes of memory are required per word in the list.

Many of the techniques described above can be used to build perfect hashing functions for this application. These algorithms cover a wide range of complexities, though the simpler algorithms often have significant disadvantages. In this section a simple algorithm will be described and used to demonstrate perfect hashing, and also to demonstrate the six issues described above.

An extremely simple perfect hashing function for Pascal reserved words could be implemented using a 2-D array of integers, as shown in Fig. 1. The first and last characters are used as indexes into the 2-D array, and the integer at the indexed location points to the word in the word list (normally the word list would contain information needed by the compiler to process/compile the given reserved word).

In Fig. 1, a list of seven Pascal reserved words has been hashed. To look up the word AND using this arrangement, the first letter, A, is used as the row index into the array, while the last letter, D, is used as the column index. At the location addressed by A and D is the value 1, which is used to access the record referenced by the word AND in a word list. As can be seen, the retrieval time for any of the words in the word list is constant, regardless of the number of words in the array, giving this perfect hashing function an $O(1)$ retrieval time. When using this algorithm on a very large word list, the array would be stored in primary memory and the word records would be stored on disk, allowing the word records to be accessed with a single disk probe.

The job of the perfect hashing algorithm is to build the 2-D array (a one-time operation). The perfect hashing function (used many times) would use this 2-D array along with the function shown in (7).

$$\text{Hash address} = \text{array[first letter, last letter]}$$ (7)

In (7), first letter and last letter are the first and last letter of the word being hashed. This is an extremely efficient retrieval function compared with other available algorithms.

This perfect hashing algorithm forms ordered minimal perfect hashing functions. The word list contains no gaps between the words in the list—they are all contiguous. The word list shown in Fig. 1 can also be ordered in any way.

This particular perfect hashing algorithm would also have an $O(n)$ build time. When the perfect hashing function is being built, a single value is placed in the 2-D array for each word in the word list.

The memory usage for this perfect hashing function is very inefficient because many entries in the 2-D array are empty. In the example shown in Fig. 1, 669 of the 676 entries in the 2-D array are unused (about 1% memory efficiency). For the seven words handled by this perfect hashing function, 194 bytes of memory are required per word in the list.

This algorithm is prone to pattern collisions. For example, the word CUT cannot be added to the list because it collides with CONST. Table II shows the probability of pattern collisions based on the number of words in the word list. Knuth [17] calls this phenomenon the birthday paradox [23]. For example, in Fig. 1 the array contains 676 positions. If two random words are to be placed into the array, the probability of them not colliding at the same location is $675/676 = 99.85\%$. Therefore, the probability of collision is 0.15\%. If three random words are placed in the array, then the probability of no collision decreases to $(674/676) \times (675/676) = 99.55\%$. Therefore the probability of collision in the array containing three words is 0.45\%.

From this example, it can be seen that the perfect hashing algorithm described has certain advantages and disadvantages. The advantages include the ability of the algorithm to form ordered minimal perfect hashing functions in $O(n)$ time. The perfect hashing functions formed are also extremely efficient. The disadvantages include extreme memory inefficiency and a high sensitivity to pattern collisions. The memory efficiency disadvantage would tend to rule out the perfect hashing function formed by this algorithm.

### TABLE II

<table>
<thead>
<tr>
<th>Number of Words</th>
<th>Probability of Collision</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0.15%</td>
</tr>
<tr>
<td>3</td>
<td>0.45%</td>
</tr>
<tr>
<td>10</td>
<td>7%</td>
</tr>
<tr>
<td>20</td>
<td>26%</td>
</tr>
<tr>
<td>30</td>
<td>50%</td>
</tr>
<tr>
<td>40</td>
<td>71%</td>
</tr>
<tr>
<td>50</td>
<td>85%</td>
</tr>
<tr>
<td>60</td>
<td>93%</td>
</tr>
<tr>
<td>70</td>
<td>97%</td>
</tr>
</tbody>
</table>

V. THE TRIE DATA STRUCTURE

In any set of words to be perfectly hashed, no two words have exactly the same characters in all positions. This fact is used in the trie data structure [25, 11] to allow word lists of any size to be accessed.

To retrieve a word from a trie data structure, each letter in the word (starting with the first) is examined until the word can be uniquely differentiated from others in the list of words known to the trie. For example, if the word DOG is known to the trie, and if DOG is the only word in the list that begins with a D, then when trying to retrieve the word DOG using the trie, only the first letter must be examined.

Fig. 2 shows an array-based trie for a word list containing the words CAT, COP, CORN, CORK, DOG, FLAG, and FROG. To access the word DOG from the trie, the first letter of the word (D) is used as an index into the first column of the array (using the first letter of the word to be retrieved to index the first column of the trie is the starting point for any
retrieval). The value 5 found at location D of the first column indicates that the word DOG can be found in location 5 of the word list. Because DOG is the only word in the list starting with D, the first letter is sufficient to differentiate it from all other words in the list.

To access the word FLAG from the trie, the first letter of the word is used as an index into the first column of the array. The value -5 is found. The negative sign is a flag used to indicate that the first letter is not sufficient to differentiate the word, and that the second letter should be used as an index into another column, in this case column 5, of the array. At location L in column 5, the value 6 is found. This is the location of the word FLAG in the word list.

Words that are prefixes to other words (e.g., cat is a prefix of cats) are handled in a trie by the addition of an "end of word" character to the character set (see end in Fig. 2). The end character, because it is unique in the character set, can be used to distinguish words that have the same prefix by appending it to the end of the shorter word.

The trie data structure shown in Fig. 2 would work well as a perfect hashing algorithm because it completely eliminates the pattern collision problem, and also because it produces minimal word lists that can be ordered in any way (the list shown is alphabetical). If the trie array were stored in memory, the CPU time needed to index a word through the trie would actually be less than that of other perfect hashing functions currently in use for handling large word sets [22], [9]. For example, to look up the word CORN using Sager’s algorithm [22] (or Fox’s improvement of it [9]) requires that seven characters be indexed from the word, along with 11 additions, three MOD operations, and three array indexings. The worst-case situation for a memory based trie using the word CORN requires that five letters be indexed from the word, and that the trie array be indexed five times. Retrieval time from the memory-based trie array is superior to retrieval using Sager’s minicycle algorithm. It is again important to note that time spent manipulating in-memory tables is considered insignificant compared to the time required to do a disk probe. A long word such as electroencephalogram could require a number of in-memory operations to generate the hash address using either Sager’s algorithm or the trie. This does not make the algorithm $O(n)$ or $O(\text{maximum word length})$, however, because only one disk probe is required to access the record on disk as a result of these calculations. It is the number of disk accesses that determines the order in any hashing algorithm.

The importance of a single disk probe cannot be overstressed, especially on slow devices such as CD-ROM’s.

VI. COMPRESSION OF THE TRIE ARRAY

If the trie array were stored in memory, it could provide a simple scheme for perfect hashing that is superior in terms of retrieval CPU time to other perfect hashing algorithms [22], [9]. Unfortunately, the memory utilization in the trie array shown in Fig. 2 is only 8%, and this inefficiency makes the algorithm impractical in most cases involving large word sets.

One way to make the use of a trie practical for perfect hashing would be to compress the trie array to improve its memory utilization. A simple and efficient $O(r^2)$ algorithm described in [4] as the SMP algorithm and in [24] as the double displacement method can be used here to compress the trie.

The array to be compressed is separated into its individual columns (or rows). The columns are rearranged based on the number of elements in each column so that the column containing the most elements is at the top of the sorted list. The columns are then copied into a 1-D array, starting with the column containing the most elements. The relationship between the elements in each column is preserved as the column is copied into the 1-D array. A column lookup table (CLT) is maintained to remember the starting location of each column copied into the 1-D array. Each column is placed into the 1-D array at the first available position that causes no collisions between filled elements of the new column being placed into the 1-D array and the already filled elements of the 1-D array. A pseudocode version of this algorithm, as it would be applied to tries, is shown in Fig. 3.

Applying this sparse array packing algorithm to the trie array shown in Fig. 2 would result in the 1-D array shown in Fig. 4.

The CLT in Fig. 4 indicates the starting location of each column that has been packed into the 1-D array. To retrieve the word FLAG using the setup shown in Fig. 4, the first
Fig. 5. The packed trie array. To find column 1 in the packed trie array, the value is added to 6 (since $F$ is the 6th letter in the alphabet) to yield $6$. Since 6 is positive, it is used as an index into the word list of Fig. 2, where the word $FLA G$ is found.

The packed trie's memory utilization can be improved to 100% using the MSMP sparse array packing technique shown in [4]. This algorithm adds a MOD function to retrievals in the 1-D array to eliminate unused gaps in the array. This extra step is not necessary, however. On large word lists (any list of over 100 words) the packing algorithm shown in Figs. 4 through 6 yields utilizations of 100% in every test case that has been attempted.

This phenomenon occurs because of the way the 2-D trie is constructed and packed into the 1-D array. For example, on a 3000-word list used in testing, the 2-D trie had 2122 columns. Each column contains 128 values (one value for each of the 128 ASCII codes). In this 2-D array, 5121 positions contain values. The 1-D array will therefore contain 5121 positions if memory utilization is achieved. It can be seen that each column placed into the 1-D array is very small in relation to the total 1-D array size.

 Examination of the composition of the 2-D trie shows that a majority of the columns contain only one or two values: 911 columns contain two values, while 680 columns contain one value. This column distribution occurs because many words must be separated from their plural forms in the trie. For example, if the words $MARSHA, MARSHALL, and MARSHALLS$ occur in the list, two single value columns (each containing an $L$) will be created in the trie to get to the point where $MARSHALL$ and $MARSHALLS$ can be distinguished. Then a two value column (containing $S$ and an end-of-word mark) will be created to cause the actual distinguishing between $MARSHALL$ and $MARSHALLS$. The columns containing two values pack very densely. The columns that contain only one value can be placed anywhere in the 1-D array, so they are used by the algorithm to fill all unused gaps left in the 1-D array. The result has been 100% memory utilization in the 1-D array for every test case that has been attempted. The general organization of a trie does not guarantee 100% utilization of memory when the trie is packed, but it makes it extremely likely.

VII. RESULTS

Table III shows the amount of time and space required to form ordered minimal perfect hashing functions using the compressed trie algorithm described above. The final entry in this table shows the performance of the algorithm when it is used to form an ordered minimal perfect hashing function for the entire Unix word list (including such words as $10th,$ $I'll,$ and $electroencephalography$) normally found in the file /usr/dict/words. This word list contains letters (both upper and lower case), digits, and punctuation. To handle the entire list, the columns of the trie were increased to 128 locations to accommodate the entire ASCII character set.
In Table III, word lists of the size indicated were taken from the standard Unix word list (e.g., the first 1000 words were taken for the 1000-word test). The Formation Time column indicates the amount of time taken to form the 2-D trie in seconds. The Packing Time column indicates the amount of time required to form the packed 1-D array. Total time is the sum of formation time and packing time. Columns indicates the number of columns in the 2-D array formed, while Values indicates the number of filled values in the 2-D array. This column also indicates the size of the 1-D array formed, because in all cases the 2-D array packed perfectly into the 1-D array, with no unused values present in the 1-D array. This table was formed using one node of a Sequent Balance 8000 (each node uses a 12-MHz 32032 processor, which is roughly equivalent to a Macintosh II computer (a 68020 at 16 Mhz)).

Determining the computational complexity of the trie packing algorithm is straightforward. It can be seen in Table III that the size of the 2-D trie, both in terms of the number of columns and in terms of the number of values held in the trie, is directly proportional to the number of words. Call the total number of elements held in the 2-D trie \( T \), and assume in the worst case that there are \( r \) columns in the 2-D trie, each containing 1 element. Because the 2-D trie contains \( r \) values, a minimally packed 1-D array containing the contents of this 2-D trie will also require \( r \) elements. As each column is packed, every element in the column must be examined to determine if it collides with any existing values in the 1-D array. Also, in the worst case, there are \( r \) possible positions for each column to occupy in the 1-D array. Since each column can be considered to contain a single element, and since in the worst case \( r \) positions for this column in the 1-D array must be examined, \( r \) comparisons must be made. This same operation must be performed for all \( r \) columns in the 2-D trie. Therefore, \( r^2 \) comparisons must be performed to pack the 2-D trie. The algorithm is \( O(r^2) \) in the worst case. The \( O(r^2) \) nature of this algorithm can also be seen in the Total Time column of Table III.

The time required to construct the trie to be packed is \( O(n) \), as can be seen empirically in Table III. This is a one-time cost, like the packing of the trie. For more information on trie construction, see [25].

Table IV compares the new algorithm with other current minimal perfect hashing algorithms. It can be seen that the new algorithm can be used to create perfect hashing functions for extremely large word lists. It should also be noted that the packed trie technique is the only algorithm in Table IV that forms an ordered perfect hashing function, and it is also the only algorithm that is immune to pattern collisions. In Table IV, the Space column indicates the amount of memory space (in bytes) required to hold the table used by the given algorithm, per item stored in the item list. The space requirement for the packed trie algorithm was determined by dividing the total number of elements in the packed trie (at 2 bytes per element) by the total number of words recognizable by the packed trie for the 24481-word trie (see Table III): 41442 + 2/24481 = 3.4. The elements in the packed trie will be stored in memory and used by the perfect hashing function during retrieval. This process is typical of all of the perfect hashing functions shown in Table IV, including Cichelli’s algorithm and Sager’s algorithm.

It is also important to recognize that this table of elements is not a hash table. It is a unique table of values formed for the data set, and it allows every item in the data set to be accessed on the disk using a single probe of the storage medium. Normal hashing algorithms would have poor performance compared to the implementation shown here.

The problem involved with using a normal hash function in a perfect hashing application can be clarified with an example. Double hashing is a well understood and effective hashing algorithm that could be used to allow word searches equivalent to the perfect hashing function’s. One of two techniques might be used. In the first, a table of pointers is held in memory, as shown in Fig. 7. To look up a word, the word is hashed and the pointer value held at the hash location in memory is used to probe the disk. This approach would yield an in-memory table of the same size as the trie, but because of collisions in the hash table multiple disk probes would be required for retrieval of some data items.

The second way to use double hashing would be to place all of the keys in an in-memory table so that multiple disk probes are not required. This approach is shown in Fig. 8. Double hashing is used to find the word in the table. Once the word is found, the associated pointer is used to access the word on the disk in one probe. While this would yield the desired disk

<table>
<thead>
<tr>
<th># Words</th>
<th>Formation Time (s)</th>
<th>Packing Time (s)</th>
<th>Total Time (s)</th>
<th># Columns</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>30</td>
<td>31</td>
<td>61</td>
<td>795</td>
<td>1794</td>
</tr>
<tr>
<td>1500</td>
<td>45</td>
<td>52</td>
<td>97</td>
<td>1233</td>
<td>2722</td>
</tr>
<tr>
<td>3000</td>
<td>88</td>
<td>128</td>
<td>216</td>
<td>2122</td>
<td>5121</td>
</tr>
<tr>
<td>6000</td>
<td>183</td>
<td>360</td>
<td>543</td>
<td>4250</td>
<td>10249</td>
</tr>
<tr>
<td>12000</td>
<td>385</td>
<td>1141</td>
<td>1526</td>
<td>8592</td>
<td>20591</td>
</tr>
<tr>
<td>24481</td>
<td>789</td>
<td>4231</td>
<td>5020</td>
<td>16962</td>
<td>41442</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Algorithm Name</th>
<th>Reference</th>
<th>Build Order</th>
<th>List Size</th>
<th>CPU Time (s)</th>
<th>Machine</th>
<th>Space (bytes/entry)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cichelli</td>
<td>[6]</td>
<td>O(r^2)</td>
<td>40</td>
<td>35</td>
<td>IBM</td>
<td>0.65</td>
</tr>
<tr>
<td>Karplus</td>
<td>[15]</td>
<td>O(r^1.5)^*</td>
<td>667</td>
<td>150</td>
<td>Vax</td>
<td>N.A.</td>
</tr>
<tr>
<td>Sager</td>
<td>[22]</td>
<td>O(r^3)^*</td>
<td>256</td>
<td>45</td>
<td>IBM</td>
<td>4.0</td>
</tr>
<tr>
<td>Fox</td>
<td>[9]</td>
<td>O(r^3)</td>
<td>1000</td>
<td>600</td>
<td>Mac II</td>
<td>4.0</td>
</tr>
<tr>
<td>Brain</td>
<td>[3]</td>
<td>O(r^2)^*</td>
<td>1696</td>
<td>733</td>
<td>PS/2/250</td>
<td>2.4</td>
</tr>
<tr>
<td>MSMP</td>
<td>[4]</td>
<td>O(r^2)</td>
<td>5000</td>
<td>283</td>
<td>PS/2/70</td>
<td>2.0</td>
</tr>
<tr>
<td>Packed Trie</td>
<td></td>
<td></td>
<td>24481</td>
<td>5020</td>
<td>SB 8000</td>
<td>3.4</td>
</tr>
</tbody>
</table>

N.A. = Not available.

*Derived from experimental results rather than theoretical analysis.
**These times are included for completeness rather than comparison purposes, although the machines listed are all roughly equivalent to one another in performance. The times demonstrate that each of the algorithms listed can complete its task in a reasonable amount of time.
A comparison of the way a perfect hashing function and a normal hash function use an in-memory table.

Double hashing using an in-memory table

Perfect Hashing Algorithm's Use of a Table

Normal Hashing Algorithm's Use of a Table

EXTENSIONS OF THE NEW ALGORITHM

The new algorithm offers a number of advantages over other current algorithms as long as the packed trie can be held in memory. On extremely long word lists (e.g., a 100,000-word list found in a medical dictionary located on a CD-ROM), however, it may be infeasible to store the entire packed trie in memory. This would cause multiple reads to be performed on the packed trie stored on disk (a maximum of five reads, as already discussed in the case of the word CORN), and these multiple reads will severely degrade performance.

To solve this problem, the word list can be segmented as discussed in [9]. For example, a 100,000-word list could be subdivided into 26 separate lists using the first character of each word to determine the contents of each sublist (similarly, the first two characters could be used to subdivide the lists into 676 sublist, etc.). To retrieve a word, the word would first be examined to determine which subtrie is appropriate, and that subtrie would be loaded into memory in a single read operation. The normal retrieval process would then continue using the subtrie stored in memory. Since individual sectors on CD-ROM's are extremely large (2 Kbytes [27]), a significant number of words can be stored in each subtrie.

IX. CONCLUSION

A packed trie data structure can be used to create extremely fast and space-efficient ordered minimal perfect hashing functions for extremely large word lists. The performance of the algorithm, both in terms of access and creation times, is superior to other perfect hashing functions currently available.

REFERENCES


Marshall Brain received the B.S. degree in computer engineering from Rensselaer Polytechnic Institute, Troy, NY, and the M.S. degree in computer science from North Carolina State University, Raleigh, NC.

He is a member of the Academy of Outstanding Teachers at North Carolina State University. He currently works for Interface Technologies, Inc. in Wake Forest, NC.

Mr. Brain is the author of Motif Programming: The Essentials . . . and More (Digital Press). He is the lead author Prentice Hall's five-book Bridge series on Windows NT.

Alan L. Tharp received the B.S. degree in science engineering and the M.S. and Ph.D. degrees in computer science from Northwestern University, Evanston, IL.

He is currently Head of the Computer Science Department at North Carolina State University, Raleigh, NC. His research interests include file and data structures, database management systems, and human–computer interfaces.

Dr. Tharp is a member of the Association for Computing Machinery. He is also the author of File Organization and Processing (Wiley, 1988).