GPERF
A Perfect Hash Function Generator

Douglas C. Schmidt
schmidt@cs.wustl.edu
http://www.cs.wustl.edu/~schmidt/
Department of Computer Science
Washington University, St. Louis 63130

This paper appeared in the C++ Report, November/December, 1998. An earlier version of this paper appeared in the 2^{nd} USENIX C++ Conference in San Francisco, California, April 1990.

Abstract

`gperf` is a "software-tool generating-tool" designed to automate the generation of perfect hash functions. This paper describes the features, algorithms, and object-oriented design and implementation strategies incorporated in `gperf`. It also presents the results from an empirical comparison between `gperf`-generated recognizers and other popular techniques for reserved word lookup. `gperf` is distributed with the GNU libg++ library and is used to generate the keyword recognizers for the GNU C/C++ compilers and the TAO CORBA IDL compiler.

1 Introduction

Perfect hash functions are a time and space efficient implementation of *static search sets*. A static search set is an abstract data type (ADT) with operations *initialize*, *insert*, and *retrieve*. Static search sets are common in system software applications. Typical static search sets include compiler and interpreter reserved words, assembler instruction mnemonics, shell interpreter built-in commands, and CORBA IDL compilers. Search set elements are called *keywords*. Keywords are inserted into the set once, usually off-line at compile-time.

`gperf` is a freely available perfect hash function generator written in C++ that automatically constructs perfect hash functions from a user-supplied list of keywords. It was designed in the spirit of utilities like `lex` [1] and `yacc` [2] to remove the drudgery associated with constructing time and space efficient keyword recognizers manually.

`gperf` translates an n element list of user-specified keywords, called the *keyfile*, into source code containing a k element lookup table and the following pair of functions:

- `hash` uniquely maps keywords in the *keyfile* onto the range $0..k-1$, where $k \geq n$. If $k = n$ `hash` is considered a *minimal* perfect hash function.
- `in_word_set` uses `hash` to determine whether a particular string of characters occurs in the *keyfile*, using at most one string comparison in the common case.

`gperf` is designed to run quickly for keyfiles containing several thousand keywords. `gperf` generates efficient ANSI and K&R C and C++ source code as output. It has been used to generate reserved keyword recognizers in lexical analyzers for several production and research compilers and language processing tools, including GNU C/C++ [3] and the TAO CORBA IDL compiler [4].

This paper is organized as follows: Section 2 outlines alternative static search set implementations and compares them with `gperf`-generated hash tables; Section 3 presents a sample input keyfile; Section 4 highlights design patterns and implementation strategies used to develop `gperf`; Section 5 shows the results from empirical benchmarks between `gperf`-generated recognizers and other popular techniques for reserved word lookup; Section 6 outlines the limitations with `gperf`and potential enhancements; and Section 7 presents concluding remarks.

2 Static Search Set Implementations

There are numerous implementations of static search sets. Common examples include sorted and unsorted arrays and linked lists, AVL trees, optimal binary search trees, digital search tries, deterministic finite-state automata, and various hash table schemes, such as open addressing and bucket chaining [5].

Different static search structure implementations offer trade-offs between memory utilization and search time efficiency and predictability. For example, an n element sorted array is space efficient. However, the average- and worst-case time complexity for retrieval operations using binary search on a sorted array is proportional to $O(\log n)$ [5].

In contrast, chained hash table implementations locate a table entry in constant, *i.e.*, $O(1)$, time on the average. However, hashing typically incurs additional memory overhead for link pointers and/or unused hash table buckets. In addition, hashing exhibits $O(n^2)$ worst-case performance [5].

A *minimal perfect hash function* is a static search set implementation defined by the following two properties:

The perfect property: Locating a table entry requires $O(1)$ time, *i.e.*, *at most* one string comparison is required to perform keyword recognition within the static search set.

The minimal property: The memory allocated to store the keywords is precisely large enough for the keyword set and *no larger*.

Minimal perfect hash functions provide a theoretically optimal time and space efficient solution for static search sets [5]. However, they can be hard to generate efficiently due to the extremely large search space of potential perfect hashing functions. Therefore, the following variations are often more appropriate for many practical hashing applications, especially those involving thousands of keywords:

Non-minimal perfect hash functions: These functions do not possess the minimal property since they return a range of hash values larger than the total number of keywords in the table. However, they *do* possess the perfect property since at most one string comparison is required to determine if a string is in the table. There are two reasons for generating non-minimal hash functions:

1. *Generation efficiency* – It is usually much faster to generate non-minimal perfect functions than to generate *minimal perfect* hash functions [6, 7].
2. *Run-time efficiency* – Non-minimal perfect hash functions may also execute faster than minimal ones when searching for elements that are *not* in the table because the "null" entry will be located more frequently. This situation often occurs when recognizing programming language reserved words in a compiler [8].

Near-perfect hash functions: Near-perfect hash functions do not possess the perfect property since they allow non-unique keyword hash values [9] (they may or may not possess the minimal property, however). This technique is a compromise that trades increased *generated-code-execution-time* for decreased *function-generation-time*. Near-perfect hash functions are useful when main memory is at a premium since they tend to produce much smaller lookup tables than non-minimal perfect hash functions.

```
%{
#include <stdio.h>
#include <string.h>
/* Command-line options:
   -C -p -a -n -t -o -j 1 -k 2,3
   -N is_month */
%}
struct months {
  char *name;
  int number;
  int days;
  int leap_days;
};
%%
january,          1,      31,      31
february,         2,      28,      29
march,            3,      31,      31
april,            4,      30,      30
may,              5,      31,      31
june,             6,      30,      30
july,             7,      31,      31
august,           8,      31,      31
september,        9,      30,      30
october,         10,      31,      31
november,        11,      30,      30
december,        12,      31,      31
%%
/* Auxiliary code goes here... */
#ifdef DEBUG
int main () {
  char buf[BUFSIZ];
  while (gets (buf)) {
    struct months *p = is_month (buf, strlen (buf));
    printf ("%s is%s a month\n",
            p ? p->name : buf, p ? "" : " not");
  }
}
#endif
```

Figure 1: An Example Keyfile for Months of the Year

`gperf` can generate minimal perfect, non-minimal perfect, and near-perfect hash functions, as described below.

3 Interacting with GPERF

This section explains how end-users can interact with `gperf`. By default, `gperf` reads a keyword list and optional *associated attributes* from the standard input `keyfile`. Keywords are specified as arbitrary character strings delimited by a user-specified field separator that defaults to `','`. Thus, keywords may contain spaces and any other ASCII characters. Associated attributes can be any C literals. For example, keywords in Figure 1 represent months of the year. Associated attributes in this figure correspond to fields in `struct months`. They include the number of leap year and non-leap year days in each month, as well as the months' ordinal numbers, *i.e.*, january =

1, february = 2, . . . , december = 12.

gperf's input format is similar to the UNIX utilities lex and yacc. It uses the following input format:

```
declarations and text inclusions
%%
keywords and optional attributes
%%
auxiliary code
```

A pair of consecutive % symbols in the first column separate declarations from the list of keywords and their optional attributes. C or C++ source code and comments are included verbatim into the generated output file by enclosing the text inside %{ %} delimiters, which are stripped off when the output file is generated, *e.g.*:

```
%{
#include <stdio.h>
#include <string.h>
/* Command-line options:
   -C -p -a -n -t -o -j 1 -k 2,3
   -N is_month */
%}
```

An optional user-supplied struct declaration may be placed at the end of the declaration section, just before the %% separator. This feature enables "typed attribute" initialization. For example, in Figure 1 struct months is defined to have four fields that correspond to the initializer values given for the month names and their respective associated values, *e.g.*:

```
struct months {
  char *name;
  int number;
  int days;
  int leap_days;
};
%%
```

Lines containing keywords and associated attributes appear in the *keywords and optional attributes* section of the keyfile. The first field of each line always contains the keyword itself, left-justified against the first column and without surrounding quotation marks. Additional attribute fields can follow the keyword. Attributes are separated from the keyword and from each other by field separators, and they continue up to the "end-of-line marker," which is the newline character ('\n') by default.

Attribute field values are used to initialize components of the user-supplied struct appearing at the end of the declaration section, *e.g.*:

```
january,          1,     31,     31
february,         2,     28,     29
march,            3,     31,     31
...
```

As with lex and yacc, it is legal to omit the initial declaration section entirely. In this case, the keyfile begins with the first non-comment line (lines beginning with a "#" character are treated as comments and ignored). This format style is useful for building keyword set recognizers that possess no associated attributes. For example, a perfect hash function for *frequently occurring English words* can efficiently filter out uninformative words, such as "the," "as," and "this," from consideration in a *key-word-in-context* indexing application [5].

Again, as with lex and yacc, all text in the optional third *auxiliary code* section is included verbatim into the generated output file, starting immediately after the final %% and extending to the end of the keyfile. It is the user's responsibility to ensure that the inserted code is valid C or C++. In the Figure 1 example, this auxiliary code provides a test driver that is conditionally included if the DEBUG symbol is enabled when compiling the generated C or C++ code.

4 Design and Implementation Strategies

Many articles describe perfect hashing [10, 7, 11, 12] and minimal perfect hashing algorithms [8, 13, 6, 14, 15]. Few articles, however, describe the design and implementation of a general-purpose perfect hashing generator tool in detail. This section describes the data structures, algorithms, output format, and reusable components in gperf.

gperf is written in ~4,000 lines of C++ source code. C++ was chosen as the implementation language since it supports data abstraction better than C, while maintaining C's efficiency and expressiveness [16].

gperf's three main phases for generating a perfect or near-perfect hash function are shown in Figure 2 and described below:

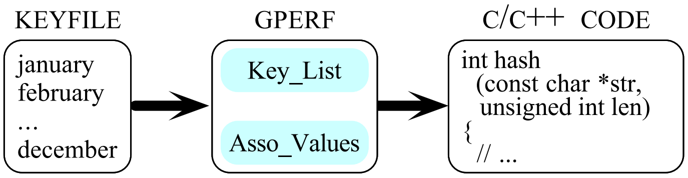

Figure 2: gperf's Processing Phases

1. Process command-line options, read keywords and attributes (the input format is described in Section 3), and initialize internal objects (described in Section 4.1).
2. Perform a non-backtracking, heuristically guided search for a perfect hash function (described in Section 4.2.1 and Section 4.2.2).

3. Generate formatted C or C++ code according to the command-line options (output format is described in Section 4.3).

The following section outlines gperf's perfect hash function generation algorithms and internal objects, examines its generated source code output, describes several reusable class components, and discusses the program's current limitations and future enhancements.

4.1 Internal Objects

gperf's implementation centers around two internal objects: the *keyword signatures* list (Key_List) and the *associated values* array (asso_values), both of which are described below.

4.1.1 The Keyword Signatures List

Every user-specified keyword and its attributes are read from the keyfile and stored in a node on a linked list, called Key_List. gperf only considers a subset of each keywords' characters while it searches for a perfect hash function. The subset is called the "keyword signature," or *keysig*.

The keysig represents the particular subset of characters used by the automatically generated recognition function to compute a keyword's hash value. Keysigs are created and cached in each node in the Key_List when the keyfile is processed initially by gperf.

4.1.2 Associated Values Array

The *associated values* array, asso_values, is an object that is closely related to keysigs. In fact, it is indexed by keysig characters. The array is constructed internally by gperf and referenced frequently while gperf searches for a perfect hash function.

During the C/C++ code generation phase of gperf, an ASCII representation of the associated array is output in the generated hash function as a static local array. This array is declared as u_int asso_values[MAX_ASCII_SIZE]. When searching for a perfect hash function, gperf repeatedly reassigns different values to certain asso_values elements specified by keysig entries. At every step during the search for the perfect hash function solution, the asso_values array's contents represent the current associated values' *configuration*.

When configured to produce minimal perfect hash functions (which is the default), gperf searches for an associated values configuration that maps all n keysigs onto non-duplicated hash values. A perfect hash function is produced when gperf finds a configuration that assigns each keysig to a unique location within the generated lookup table. The resulting perfect hash function returns an unsigned int value in the range $0..(k-1)$, where $k = (maximum\ keyword\ hash\ value + 1)$. When $k = n$ a *minimal* perfect hash function is produced. For k larger than n, the lookup table's *load factor* is $\frac{n}{k}$ ($\frac{number\ of\ keywords}{total\ table\ size}$).

A keyword's hash value is typically computed by combining the associated values of its keysig with its length.[1] By default, the hash function adds the associated value of a keyword's first index position plus the associated value of its last index position to its length, *i.e.*:

```
hash_value =
  asso_values[keyword[0]]
  + asso_values[keyword[length - 1]]
  + length;
```

Other combinations are often necessary in practice. For example, using the default hash function for C++ reserved words causes a collision between delete and double. To resolve this collision and generate a perfect hash function for C++ reserved words, an additional character must be added to the keysig, as follows:

```
hash_value =
  asso_values[keyword[0]]
  + asso_values[keyword[1]]
  + asso_values[keyword[length - 1]]
  + length;
```

Developers can control the generated hash function's contents using the "-k" option to explicitly specify the keyword index positions used as keysig elements by gperf. The default is "-k 1,$", where the '$' represents the keyword's final character.

Table 1 shows the keywords, keysigs, and hash value for each month shown in the Figure 1 keyfile. These keysigs were

Keyword	Keysig	Hash Value
january	an	3
february	be	9
march	ar	4
april	pr	2
may	ay	8
june	nu	1
july	lu	6
august	gu	7
september	ep	0
october	ct	10
november	ov	11
december	ce	5

Table 1: Keywords, Keysigs, and Hash Values for the Months Example

[1]The "-n" option instructs gperf not include the length of the keyword when computing the hash function.

produced using the -k2,3 option.

Keysigs are *multisets* since they may contain multiple occurrences of certain characters. This approach differs from other perfect hash function generation techniques [8] that only consider first/last characters + length when computing a keyword's hash value.

The hash function generated by gperf properly handles keywords shorter than a specified index position by skipping characters that exceed the keyword's length. In addition, users can instruct gperf to include *all* of a keyword's characters in its keysig via the "-k*" option.

4.2 Generating Perfect Hash Functions

This subsection describes how gperf generates perfect hash functions.

4.2.1 Main Algorithm

gperf iterates sequentially through the list of i keywords, $1 \leq i \leq n$, where n equals the total number of keywords. During each iteration gperf attempts to extend the set of uniquely hashed keywords by 1. It succeeds if the hash value computed for keyword i does not collide with the previous $i-1$ uniquely hashed keywords. Figure 3 outlines the algorithm.

for $i \leftarrow 1$ **to** n **loop**
  if hash *(i^{th} key) collides with any* hash *(1^{st} key ... $(i-1)^{st}$ key)*
  then
    modify disjoint union of associated values to resolve collisions
    based upon certain collision resolution heuristics
  end if
end loop

Figure 3: Gperf's Main Algorithm

The algorithm terminates and generates a perfect hash function when $i = n$ and no unresolved hash collisions remain. Thus, the *best-case* asymptotic time-complexity for this algorithm is linear in the number of keywords, *i.e.*, $\Omega(n)$.

4.2.2 Collision Resolution Strategies

As outlined in Figure 3, gperf attempts to resolve keyword hash collisions by incrementing certain associated values. The following discusses the strategies gperf uses to speedup collision resolution.

Disjoint union: To avoid performing unnecessary work, gperf is selective when changing associated values. In particular, it only considers characters comprising the *disjoint union* of the colliding keywords' keysigs. The disjoint union of two keysigs $\{A\}$ and $\{B\}$ is defined as $\{A \cup B\} - \{A \cap B\}$.

To illustrate the use of disjoint unions, consider the keywords january and march from Figure 1. These keywords have the keysigs ``an'' and ``ar'', respectively, as shown in Table 1. Thus, when asso_values['a'], asso_values['n'], and asso_values['r'] all equal 0, a collision will occur during gperf's execution.[2] To resolve this collision, gperf only considers changing the associated values for 'n' and/or 'r'. Changing 'a' by any increment cannot possibly resolve the collision since 'a' occurs the same number of times in each keysig.

By default, all asso_values are initialized to 0. When a collision is detected gperf increases the corresponding associated value by a "jump increment." The command-line option "-j" can be used to increase the jump increment by a fixed or random amount. In general, selecting a smaller jump increment, *e.g.*, "-j 1" decreases the size of the generated hash table, though it may increase gperf's execution time.

In the months example in Figure 1, the "-j 1" option was used. Therefore, gperf quickly resolves the collision between january and march by incrementing asso_value['n'] by 1. As shown in Table 2, this is its final value.

gperf generates a perfect hash function if increments to the associated values configuration shown in Figure 3 and described above eliminate all keyword collisions when the end of the Key_List is reached. The *worst-case* asymptotic time-complexity for this algorithm is $O(n^3 l)$, where l is the number of characters in the largest disjoint union between colliding keyword keysigs. After experimenting with gperf on many keyfiles it appears that such worst-case behavior occurs infrequently in practice.

Many perfect hash function generation algorithms [6, 7] are sensitive to the order in which keywords are considered. To mitigate the effect of ordering, gperf will optionally reorder keywords in the Key_List if the "-o" command-line option is enabled. This reordering is done in a two-stage prepass [8] before gperf invokes the main algorithm shown in Figure 3. First, the Key_List is sorted by decreasing frequency of keysig characters occurrence. The second reordering pass then reorders the Key_List so that keysigs whose values are "already determined" appear earlier in the list.

These two heuristics help to prune the search space by handling inevitable collisions early in the generation process. gperf will run faster on many keyword sets, and often decrease the perfect hash function range, if it can resolve these collisions quickly by changing the appropriate associated values. However, if the number of keywords is large and the user

[2] Note that since the "-n" option is used in the months example, the different keyword lengths are not considered in the resulting hash function.

Keysig Characters	Associated Values	Frequency of Occurrence
'a'	2	3
'b'	9	1
'c'	5	2
'e'	0	3
'g'	7	1
'l'	6	1
'n'	1	2
'o'	1	1
'p'	0	2
'r'	2	2
't'	5	1
'u'	0	3
'v'	0	1
'y'	6	1

Table 2: Associated Values and Occurrences for Keysig Characters

Search heuristics: `gperf` uses several search heuristics to reduce the time required to generate a perfect hash function. For instance, characters in the disjoint union are sorted by increasing frequency of occurrence, so that less frequently used characters are incremented before more frequently used characters. This strategy is based on the assumption that incrementing infrequently used characters *first* decreases the negative impact on keywords that are already uniquely hashed with respect to each other. Table 2 shows the associated values and frequency of occurrences for all the keysig characters in the months example.

wishes to generate a near-perfect hash function, this reordering sometimes *increases* `gperf`'s execution time. The reason for this apparent anomaly is that collisions begin earlier and frequently persist throughout the remainder of keyword processing [8, 9].

4.3 Generated Output Format

Figure 4 depicts the C code produced from the `gperf`-generated minimal perfect hash function corresponding to the keyfile depicted in Figure 1. Execution time was negligible on a Sun SPARC 20 workstation, *i.e.*, 0.0 user and 0.0 system time. The following section uses portions of this code as a working example to illustrate various aspects of `gperf`'s generated output format.

4.3.1 Generated Symbolic Constants

`gperf`'s output contains the following seven symbolic constants that summarize the results of applying the algorithm in Figure 3 to the keyfile in Figure 1:

```
enum {
  TOTAL_KEYWORDS = 12,
  MIN_WORD_LENGTH = 3,
  MAX_WORD_LENGTH = 9,
  MIN_HASH_VALUE = 0,
  MAX_HASH_VALUE = 11,
  HASH_VALUE_RANGE = 12,
  DUPLICATES = 0
};
```

`gperf` produces a *minimal perfect* hash function when HASH_VALUE_RANGE $=$ TOTAL_KEYWORDS and DUPLICATES $=$ 0. A *non-minimal perfect* hash function occurs when DUPLICATES $= 0$ and HASH_VALUE_RANGE $>$ TOTAL_KEYWORDS. Finally, a *near-perfect* hash function occurs when DUPLICATES > 0 and DUPLICATES $\ll$ TOTAL_KEYWORDS.

4.3.2 The Generated Lookup Table

By default, when `gperf` is given a keyfile as input it attempts to generate a perfect hash function that uses at most one string comparison to recognize keywords in the lookup table. `gperf` can implement the lookup table as either an array or a `switch` statement, as described below.

`asso_values` **array lookup table:** `gperf` generates an array by default, emphasizing run-time speed over minimal memory utilization. This array is called `asso_values`, as shown in the `hash` function in Figure 4. The `asso_values` array is used by the two generated functions that compute hash values and perform table lookup.

`gperf` also provides command-line options that allow developers to select trade-offs between memory size and execution time. For example, expanding the range of hash values

```
#include <stdio.h>
#include <string.h>
/* Command-line options:
   -C -p -a -n -t -o -j 1 -k 2,3
   -N is_month */
struct months {
  char *name;
  int number;
  int days;
  int leap_days;
};

enum {
  TOTAL_KEYWORDS = 12,
  MIN_WORD_LENGTH = 3,
  MAX_WORD_LENGTH  = 9,
  MIN_HASH_VALUE = 0,
  MAX_HASH_VALUE  = 11,
  HASH_VALUE_RANGE = 12,
  DUPLICATES = 0
};

static unsigned int
hash (const char *str, unsigned int len)
{
  static const unsigned char asso_values[] =
  {
    12, 12, 12, 12, 12, 12, 12, 12, 12, 12,
    12, 12, 12, 12, 12, 12, 12, 12, 12, 12,
    12, 12, 12, 12, 12, 12, 12, 12, 12, 12,
    12, 12, 12, 12, 12, 12, 12, 12, 12, 12,
    12, 12, 12, 12, 12, 12, 12, 12, 12, 12,
    12, 12, 12, 12, 12, 12, 12, 12, 12, 12,
    12, 12, 12, 12, 12, 12, 12, 12, 12, 12,
    12, 12, 12, 12, 12, 12, 12, 12, 12, 12,
    12, 12, 12, 12, 12, 12, 12, 12, 12, 12,
    12, 12, 12, 12, 12, 12, 12,  2,  9,  5,
    12,  0, 12,  7, 12, 12, 12, 12,  6, 12,
     1, 11,  0, 12,  2, 12,  5,  0,  0, 12,
    12,  6, 12, 12, 12, 12, 12, 12,
  };
  return asso_values[str[2]] + asso_values[str[1]];
}

const struct months *
is_month (const char *str, unsigned int len)
{
  static const struct months wordlist[] =
  {
    {"september", 9, 30, 30},
    {"june",      6, 30, 30},
    {"april",     4, 30, 30},
    {"january",   1, 31, 31},
    {"march",     3, 31, 31},
    {"december", 12, 31, 31},
    {"july",      7, 31, 31},
    {"august",    8, 31, 31},
    {"may",       5, 31, 31},
    {"february",  2, 28, 29},
    {"october",  10, 31, 31},
    {"november", 11, 30, 30},
  };
  if (len <= MAX_WORD_LENGTH
      && len >= MIN_WORD_LENGTH) {
    int key = hash (str, len);
    if (key <= MAX_HASH_VALUE
        && key >= MIN_HASH_VALUE) {
      char *s = wordlist[key].name;
      if (*str == *s
          && !strcmp (str + 1, s + 1))
        return &wordlist[key];
    }
  }
  return 0;
}
```

Figure 4: Minimal Perfect Hash Function Generated by `gperf`

produces a sparser lookup table. This generally yields faster keyword searches but requires additional memory.

The array-based `asso_values` scheme works best when the HASH_VALUE_RANGE is not considerably larger than the TOTAL_KEYWORDS. When there are a large number of keywords, and an even larger range of hash values, however, the `wordlist` array in `is_month` function in Figure 4 may become extremely large. Several problems arise in this case:

- The time to compile the sparsely populated array is excessive;
- The array size may be too large to store in main memory;
- A large array may lead to increased "thrashing" of virtual memory in the OS.

Switch-based lookup table: To handle the problems described above, `gperf` can also generate one or more `switch` statements to implement the lookup table. Depending on the underlying compiler's `switch` optimization capabilities, the `switch`-based method may produce smaller *and* faster code, compared with the large, sparsely filled array. Figure 5 shows how the `switch` statement code appears if the months example is generated with `gperf`'s `"-S 1"` option.

```
{
  const struct months *rw;

  switch (key)
  {
    case   0: rw = &wordlist[0];  break;
    case   1: rw = &wordlist[1];  break;
    case   2: rw = &wordlist[2];  break;
    case   3: rw = &wordlist[3];  break;
    case   4: rw = &wordlist[4];  break;
    case   5: rw = &wordlist[5];  break;
    case   6: rw = &wordlist[6];  break;
    case   7: rw = &wordlist[7];  break;
    case   8: rw = &wordlist[8];  break;
    case   9: rw = &wordlist[9];  break;
    case  10: rw = &wordlist[10]; break;
    case  11: rw = &wordlist[11]; break;
    default: return 0;
  }
  if (*str == *rw->name
      && !strcmp (str + 1, rw->name + 1))
    return rw;
  return 0;
}
```

Figure 5: The `switch`-based Lookup Table

Since the months example is somewhat contrived, the trade-off between the array and `switch` approach is not particularly obvious. However, good C++ compilers generate assembly code implementing a "binary-search-of-labels" scheme if the `switch` statement's `case` labels are sparse compared to the range between the smallest and largest `case` labels [3]. This technique can save a great deal of space by not emitting unnecessary empty array locations or jump-table slots. The exact time and space savings of this approach varies according to the underlying compiler's optimization strategy.

gperf generates source code that constructs the array or switch statement lookup table at *compile-time*. Therefore, initializing the keywords and any associated attributes requires little additional execution-time overhead when the recognizer function is run. The "initialization" is automatically performed as the program's binary image is loaded from disk into main memory.

4.3.3 The Generated Functions

gperf generates a hash function and a lookup function. By default, they are called hash and in_word_set, although a different name may be given for in_word_set using the "-N" command-line option. Both functions require two arguments, a pointer to a NUL-terminated ('\0') array of characters, const char *str, and a length parameter, unsigned int len.

The generated hash function (hash)**:** Figure 4 shows the hash function generated from the input keyfile shown in Figure 1. The command-line option "-k 2,3" was enabled for this test. This instructs hash to return an unsigned int hash value that is computed by using the ASCII values of the 2^{nd} and 3^{rd} characters from its str argument into the local static array asso_values.[3] The two resulting numbers are added to calculate str's hash value.

The asso_values array is generated by gperf using the algorithm in Section 4.1.2. This array maps the user-defined keywords onto unique hash values. All asso_values array entries with values greater than MAX_HASH_VALUE (*i.e.*, all the "12's" in the asso_values array in Figure 4) represent ASCII characters that do not occur as either the second or third characters in the months of the year. The is_month function in Figure 4 uses this information to quickly eliminate input strings that cannot possibly be month names.

Generated lookup function (in_word_set)**:** The in_word_set function is the entry point into the perfect hash lookup function. In contrast, the hash function is declared static and cannot be invoked by application programs directly. If the function's first parameter, char *str, is a valid user-define keyword, in_word_set returns a pointer to the corresponding record containing each keyword and its associated attributes; otherwise, a NULL pointer is returned.

Figure 4 shows how the in_word_set function can be renamed to is_month using the "-N" command-line option. Note how gperf checks the len parameter and resulting hash function return value against the symbolic constants for MAX_WORD_LENGTH, MIN_WORD_LENGTH, MAX_HASH_VALUE, and MIN_HASH_VALUE. This check quickly eliminates many non-month names from further consideration. If users know in advance that all input strings are valid keywords, gperf will suppress this addition checking with the "-O" option.

If gperf is instructed to generate an array-based lookup table the generated code is quite concise, *i.e.*, once it is determined that the hash value lies within the proper range the code is simply:

```
{
  char *s = wordlist[key];
  if (*s == *str
      && !strcmp (str + 1, s + 1))
    return s;
}
```

The *s == *str expression quickly detects when the computed hash value indexes into a "null" table slot since *s is the NUL character ('\0') in this case. This check is useful when searching a sparse keyword lookup table, where there is a higher probability of locating a null entry. If a null entry is located, there is no need to perform a full string comparison.

Since the months' example generates a minimal perfect hash function null enties never appear. The check is still useful, however, since it avoids calling the string comparison function when the str's first letter does not match any of the keywords in the lookup table.

4.4 Reusable Components and Patterns

Figure 6 illustrates the key components used in gperf's software architecture. gperf is constructed from reusable

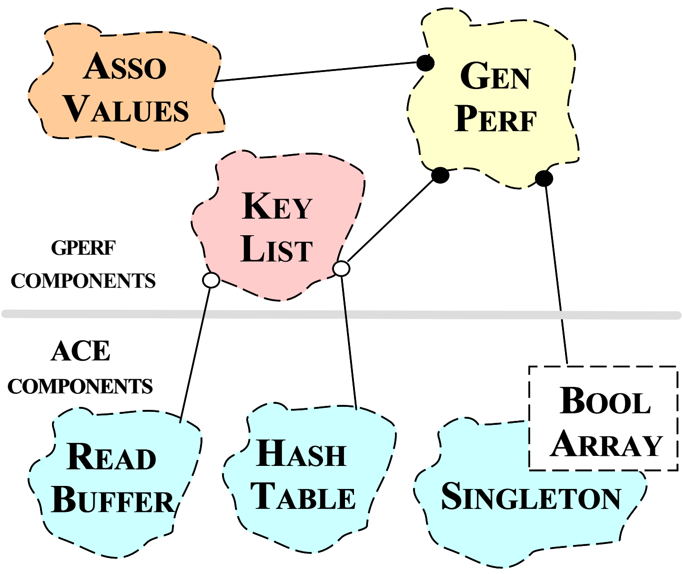

Figure 6: gperf's Software Architecture

[3]Note that C arrays start at 0, so str[1] is actually the second character.

components from the ACE framework [17]. Each component evolved "bottom-up" from special-purpose utilities into reusable software components. Several noteworthy reusable classes include the following components:

ACE_Bool_Array: Earlier versions of `gperf` were instrumented with a run-time code profiler on large input keyfiles that evoke many collisions. The results showed that `gperf` spent approximately 90 to 99 percent of its time in a single function when performing the algorithm in Figure 3. This one function, `Gen_Perf::affects_previous`, determines how changes to associated values affect previously hashed keywords. In particular, it identifies duplicate hash values that occur during program execution.

Since this function is called so frequently, it is important to minimize its execution overhead. Therefore, `gperf` employs a novel boolean array component called `ACE_Bool_Array` to expedite this process. The C++ interface for the `ACE_Bool_Array` class is depicted in Figure 7. Since only one copy is required, `BOOL_ARRAY` is `typedef`'d

```
class ACE\_Bool_Array
{
public:
  // Constructor.
  ACE\_Bool_Array (void);

  // Returns dynamic memory to free store.
 ~ACE_Bool_Array (void);

  // Allocate a k element dynamic array.
  init (u_int k);

  // Checks if <value> is a duplicate.
  int in_set (u_int value);

  // Reinitializes all set elements to FALSE.
  void reset (void);

private:
  // Current generation count.
  u_short generation_number_;

  // Dynamically allocated storage buffer.
  u_short *array_;

  // Length of dynamically allocated array.
  u_int size_;
};

// Create a Singleton.
typedef ACE_Singleton <ACE_Bool_Array,
                       ACE_Null_Mutex>
        BOOL_ARRAY;
```

Figure 7: The ACE Boolean Array Component

to be a Singleton using the `ACE_Singleton` adapter. This template automatically transforms a class into a Singleton using the Singleton and Adapter patterns [18].

The `in_set` method efficiently detects duplicate keyword hash values for a given associated values configuration. It returns non-zero if a value is already in the set and zero otherwise. Whenever a duplicate is detected, the `reset` method is called to reset all the array elements back to "empty" for ensuing iterations of the search process.

If many hash collisions occur, the `reset` method is executed frequently during the duplicate detection and elimination phase of `gperf`'s algorithm. Processing large keyfiles, *e.g.*, containing more than 1,000 keywords, tends to require a maximum hash value k that is often *much* larger than n, the total number of keywords. Due to the large range, it becomes expensive to explicitly reset all elements in `array_` back to empty, especially when the number of keywords actually checked for duplicate hash values is comparatively small.

To address this issue, `gperf` uses a pattern called *generation numbering*, which optimizes the search process by not explicitly reinitializing the entire array. Generation numbering operates as follows:

1: The `Bool_Array init` method dynamically allocates space for k `unsigned short` integers and points `array_` at the allocated memory. All k array elements in `array_` are initially assigned 0 (representing "empty") and the `generation_number_` counter is set to 1.

2: `gperf` uses the `in_set` method to detect duplicate keyword hash values. If the number stored at the `hash(keyword)` index position in `array_` is not equal to the current generation number, then that hash value is not already in the set. In this case, the current generation number is assigned immediately to the `hash(keyword)` array location, thereby marking it as a duplicate if it is referenced subsequently during this particular iteration of the search process.

3: If `array_[hash(keyword)]` is equal to the generation number, a duplicate exists and the algorithm must try to modify certain associated values to resolve the collision.

4: If a duplicate is detected, the `array_` elements are reset to empty for subsequent iterations of the search process. The `reset` method simply increments `generation_number_` by 1. The entire k array locations are only reinitialized to 0 when the generation number exceeds the range of an `unsigned short` integer, which occurs infrequently in practice.

A design strategy employed throughout `gperf`'s implementation is "first determine a concise set of operations and interfaces, then successively tune the implementation." In the case of generation numbering, this policy of optimizing performance, without compromising program clarity, decreased `gperf`'s execution-time by an average of 25 percent for large keyfiles, compared with the previous method that explicitly "zeroed out" the entire boolean array's contents on every `reset`.

ACE_Read_Buffer: Each line in gperf's input contains a single keyword followed by any optional associated attributes, ending with a newline character ('\n'). The Read_Buffer::read member function copies an arbitrarily long '\n'-terminated string of characters from the input into a dynamically allocated buffer. A recursive auxiliary function, Read_Buffer::rec_read, ensures only one call is made to the new opeator for each input line read, *i.e.*, there is no need to reallocate and resize buffers dynamically. This class has been incorporated into the GNU libg++ **stream** library [19] and the ACE network programming tookit [17].

ACE_Hash_Table: This class provides a search set implemented via double hashing [5]. During program initialization gperf uses an instance of this class to detect keyfile entries that are guaranteed to produce duplicate hash values. These duplicates occur whenever keywords possess both identical keysigs and identical lengths, *e.g.*, the double and delete collision described in Section 4.1.2. Unless the user specifies that a near-perfect hash function is desired, attempting to generate a perfect hash function for keywords with duplicate keysigs and identical lengths is an exercise in futility!

5 Empirical Results

Tool-generated recognizers are useful from a software engineering perspective since they reduce development time and decrease the likelyhood of development errors. However, they are not necessarily advantageous for production applications unless the resulting executable code speed is competitive with typical alternative implementations. In fact, it has been argued that there are *no* circumstances where perfect hashing proves worthwhile, compared with other common static search set methods [20].

To compare the efficacy of the gperf-generated perfect hash functions against other common static search set implementations, seven test programs were developed and executed on six large input files. Each test program implemented the same function: a recognizer for the reserved words in GNU g++ . The function returns 1 if a given input string is identified as a reserved word and 0 otherwise.

The seven test programs are described below. They are listed by increasing order of execution time, as shown in Table 3. The input files used for the test programs are described in Table 4. Table 5 shows the number of bytes for each test program's compiled object file, listed by increasing size (both patricia.o and chash.o use dynamic memory, so their overall memory usage depends upon the underlying free store mechanism).

Input File	Identifiers	Keywords	Total
ET++.in	624,156	350,466	974,622
NIH.in	209,488	181,919	391,407
g++.in	278,319	88,169	366,488
idraw.in	146,881	74,744	221,625
cfront.in	98,335	51,235	149,570
libg++.in	69,375	50,656	120,031

Table 4: Total Identifiers and Keywords for Each Input File

Object File	Byte Count				
	text	data	bss	dynamic	total
control.o	88	0	0	0	88
binary.o	1,008	288	0	0	1,296
gperf.o	2,672	0	0	0	2,672
chash.o	1,608	304	8	1,704	3,624
patricia.o	3,936	0	0	2,272	6,208
comp-flex.o	7,920	56	16,440	0	24,416
trie.o	79,472	0	0	0	79,472
flex.o	3,264	98,104	16,440	0	117,808

Table 5: Size of Object Files in Bytes

trie.exe: a program based upon an automatically generated table-driven search trie created by the **trie-gen** utility included with the GNU libg++ distribution.

flex.exe: a flex-generated recognizer created with the "-f" (no table compaction) option. Note that both the flex.exe and trie.exe are uncompacted, deterministic finite automata (DFA)-based recognizers. Not using compaction maximizes speed in the generated recognizer, at the expense of much larger tables. For example, the uncompacted flex.exe program is almost 5 times larger than the compacted comp-flex.exe program, *i.e.*, 117,808 bytes versus 24,416 bytes.

gperf.exe: a gperf-generated recognizer created with the "-a -D -S 1 -k 1,$" options. These options mean "generate ANSI C prototypes ("-a"), handle duplicate keywords ("-D"), via a single switch statement ("-S 1"), and make the keysig be the first and last character of each keyword."

chash.exe: a dynamic chained hash table lookup function similar to the one that recognizes reserved words for AT&T's cfront 3.0 C++ compiler. The table's load factor is 0.39, the same as it is in cfront 3.0.

patricia.exe: a PATRICIA trie recognizer, where PATRICIA stands for "Practical Algorithm to Retrieve Information Coded in Alphanumeric." A complete PATRICA trie implementation is available in the GNU libg++ class library distribution [19].

binary.exe: a carefully coded binary search function that minimizes the number of complete string comparisons.

comp-flex.exe: a flex-generated recognizer created with the default "-cem" options, providing the highest degree

Executable Program	Input File					
	ET++.in	NIH.in	g++.in	idraw.in	cfront.in	libg++.in
control.exe	38.8 | 1.00	15.4 | 1.00	15.2 | 1.00	8.9 | 1.00	5.7 | 1.00	4.5 | 1.00
trie.exe	59.1 | 1.52	23.8 | 1.54	23.8 | 1.56	13.7 | 1.53	8.6 | 1.50	7.0 | 1.55
flex.exe	60.5 | 1.55	23.9 | 1.55	23.9 | 1.57	13.8 | 1.55	8.9 | 1.56	7.1 | 1.57
gperf.exe	64.6 | 1.66	26.0 | 1.68	25.1 | 1.65	14.6 | 1.64	9.7 | 1.70	7.7 | 1.71
chash.exe	69.2 | 1.78	27.5 | 1.78	27.1 | 1.78	15.8 | 1.77	10.1 | 1.77	8.2 | 1.82
patricia.exe	71.7 | 1.84	28.9 | 1.87	27.8 | 1.82	16.3 | 1.83	10.8 | 1.89	8.7 | 1.93
binary.exe	72.5 | 1.86	29.3 | 1.90	28.5 | 1.87	16.4 | 1.84	10.8 | 1.89	8.8 | 1.95
comp-flex.exe	80.1 | 2.06	31.0 | 2.01	32.6 | 2.14	18.2 | 2.04	11.6 | 2.03	9.2 | 2.04

Table 3: Raw and Normalized CPU Processing Time

of table compression. Note the obvious time/space trade-off between the uncompacted `flex.exe` (which is faster and larger) and the compacted `comp-flex.exe` (which is smaller and much slower).

In addition to these seven test programs, a simple C++ program called `control.exe` measures and controls for I/O overhead, *i.e.*:

```
int main (void) {
  char buf[BUFSIZ];

  while (gets (buf))
    printf ("%s", buf);
}
```

All of the above reserved word recognizer programs were compiled by the GNU g++ 2.7.2 compiler with the `"-O2 -finline-functions"` options enabled. They were then tested on an otherwise idle SPARCstation 20 model 712 with 128 megabytes of RAM.

All six input files used for the tests contained a large number of words, both user-defined identifiers and g++ reserved words, organized with one word per line. This formate was automatically created by running the UNIX command `"tr -cs A-Za-z_ '\012'"` on the preprocessed source code for several large C++ systems, including the ET++ windowing toolkit (`ET++.in`), the NIH class library (`NIH.in`), the GNU g++ 2.7.2 C++ compiler (`g++.in`), the `idraw` figure drawing utility from the InterViews 2.6 distribution (`idraw.in`), the AT&T `cfront` 3.0 C++ compiler (`cfront.in`), and the GNU libg++ 2.8 C++ class library (`libg++.in`). Table 4 shows the relative number of identifiers and keywords for the test input files.

Table 3 depicts the amount of time each search set implementation spent executing the test programs, listed by increasing execution time. The first number in each column represents the user-time CPU seconds for each recognizer. The second number is "normalized execution time," *i.e.*, the ratio of user-time CPU seconds divided by the `control.exe` program execution time. The normalized execution time for each technique is very consistent across the input test file suite, illustrating that the timing results are representative for different source code inputs.

Several conclusions result from these empirical benchmarks:

Time/space tradeoffs are common: The uncompacted, DFA-based trie (`trie.exe`) and flex (`flex.exe`) implementations are both the fastest and the largest implementations, illustrating the time/space trade-off dichotomy. Applications where saving time is more important than conserving space may benefit from these approaches.

gperf can provide the best of both worlds: While the `trie.exe` and `flex.exe` recognizers allow programmers to trade-off space for time, the `gperf`-generated perfect hash function `gperf.exe` is comparatively time *and* space efficient. Empirical support for this claim can be calculated from the data for the programs that did not allocate dynamic memory, *i.e.*, `trie.exe`, `flex.exe`, `gperf.exe`, `binary.exe`, and `comp-flex.exe`. The number of identifiers scanned per-second, per-byte of executable program overhead was 5.6 for `gperf.exe`, but less than 1.0 for `trie.exe`, `flex.exe`, and `comp-flex.exe`.

Since `gperf` generates a stand-alone recognizer, it is easily incorporated into an otherwise hand-coded lexical analyzer, such as the ones found in the GNU C and GNU C++ compiler. It is more difficult, on the other hand, to partially integrate `flex` or `lex` into a lexical analyzer since they are generally used in an "all or nothing" fashion. Furthermore, neither `flex` nor `lex` are capable of generating recognizers for extremely large keyfiles because the size of the state machine is too big for their internal DFA state tables.

6 Current Limitations and Future Work

`gperf` has been freely distributed for many years along with the GNU libg++ library and the ACE network programming toolkit at www.cs.wustl.edu/˜schmidt/ACE.html. Although gperf has proven to be quite useful in practice, there are several limitations. This section describes the tradeoffs and compromises with its current algorithms and outlines how it can be improved. Since gperf is open source software, however, it is straightforward to add enhancements and extensions.

6.1 Tradeoffs and Compromises

Several other hash function generation algorithms utilize some form of backtracking when searching for a perfect or minimal perfect solution [6, 8, 9]. For example, Cichelli's [8] algorithm recursively attempts to find an associated values configuration that uniquely maps all n keywords to distinct integers in the range $1..n$. In his scheme, the algorithm "backs up" if computing the current keyword's hash value exceeds the minimal perfect table size constraint at any point during program execution. Cichelli's algorithm then proceeds by undoing selected hash table entries, reassigning different associated values, and continuing to search for a solution.

Unfortunately, the exponential growth rate associated with the backtracking search process is simply too time consuming for large keyfiles. Even "intelligently-guided" exhaustive search quickly becomes impractical for more than several hundred keywords.

To simplify the algorithm in Figure 3, and to improve average-case performance, gperf does not backtrack when keyword hash collisions occur. Thus, gperf may process the entire keyfile input, *without* finding a unique associated values configuration for every keyword, even if one exists. If a unique configuration is not found, users have two choices:

1. They can run gperf again, enabling different options in search of a perfect hash function; or
2. They can *guarantee* a solution by instructing gperf to generate an *near-perfect* hash function.

Near-perfect hash functions permit gperf to operate on keyword sets that it otherwise could not handle, *e.g.*, if the keyfile contains duplicates or there are a very large number of keywords. Although the resulting hash function is no longer "perfect," it handles keyword membership queries efficiently since only a small number of duplicates usually remain.[4]

Both duplicate keyword entries and unresolved keyword collisions are handled by generalizing the switch-based scheme described in Section 3. gperf treats duplicate keywords as members of an *equivalence class* and generates switch statement code containing cascading if-else comparisons within a case label to handle non-unique keyword hash values.

For example, if gperf is run with the default keysig selection command-line option "-k 1,$" on a keyfile containing C++ reserved words, a hash collision occurs between the delete and double keywords, thereby preventing a perfect hash function. Using the "-D" option produces a near-perfect hash function, that allows at most one string comparison for all keywords except double, which is recognized after two comparisons. Figure 8 shows the relevant fragment of the generated near-perfect hash function code.

```
{
  char *rw;
  ...
  switch (hash (str, len)) {
  ...
  case   46:
    rw = "delete";
    if (*str == *rw
        && !strcmp (str + 1, rw + 1, len - 1))
      return rw;
    rw = "double";
    if (*str == *rw
        && !strcmp (str + 1, rw + 1, len - 1))
      return rw;
    return 0;
  case   47:
    rw = "default"; break;
  case   49:
    rw = "void"; break;
  ...
  }
  if (*str == *rw
      && !strcmp (str + 1, rw + 1, len - 1))
    return rw;
  return 0;
}
```

Figure 8: The Near-Perfect Lookup Table Fragment

A simple linear search is performed on duplicate keywords that hash to the same location. Linear search is effective since most keywords still require only one string comparison. Support for duplicate hash values is useful in several circumstances, such as large input keyfiles (*e.g.*, dictionaries), highly similar keyword sets (*e.g.*, assembler instruction mnemonics), and secondary keys. In the latter case, if the primary keywords are distinguishable only via secondary key comparisons, the user may edit the generated code by hand or via an automated script to completely disambiguate the search key.

6.2 Enhancements and Extensions

Fully automating the perfect hash function generation process is gperf's most significant unfinished extension. One approach is to replace gperf's current algorithm with more exhaustive approaches [9, 7]. Due to gperf's object-oriented program design, such modifications will not disrupt the overall program structure. The perfect hash function generation mod-

[4] The exact number depends on the keyword set and the command-line options.

ule, `class Gen_Perf`, is independent from other program components; it represents only about 10 percent of `gperf`'s overall lines of source code.

A more comprehensive, albeit computationally expensive, approach could switch over to a backtracking strategy when the initial, computationally less expensive, non-backtracking first pass fails to generate a perfect hash function. For many common uses, where the search sets are relatively small, the program will run successfully without incurring backtracking overhead. In practice, the utility of these proposed modifications remains an open question.

Another potentially worthwhile feature is enhancing `gperf` to automatically select the keyword index positions. This would assist users in generating time or space efficient hash functions quickly and easily. Currently, the user must use the default behavior or explicitly select these positions via command-line arguments. Finally, `gperf`'s output functions can be extended to generate code for other languages, *e.g.*, Java, Ada, Smalltalk, Module 3, Eiffel, etc.

7 Concluding Remarks

`gperf` was originally designed to automate the construction of keyword recognizers for compilers and interpreter reserved word sets. The various features described in this paper enable it to achieve its goal, as evidenced by its use in the GNU compilers. In addition, `gperf` has been used in the following applications:

- The TAO CORBA IDL compiler [4] uses `gperf` to generate the operation dispatching tables [21] used by server-side skeletons.
- A hash function for 15,400 "Medical Subject Headings" used to index journal article citations in MEDLINE, a large bibliographic database of the biomedical literature maintained by the National Library of Medicine. Generating this hash function takes approximately 10 minutes of CPU time on a SPARC 20 workstation.
- The GNU `indent` C code reformatting program, where the inclusion of perfect hashing sped up the program by an average of 10 percent.
- A public domain program converting double precision FORTRAN source code to/from single precision uses `gperf` to modify function names that depend on the types of their arguments, *e.g.*, replacing `sgefa` with `dgefa` in the LINPACK benchmark. Each name corresponding to a function is recognized via `gperf` and substituted with the version for the appropriate precision.
- A speech synthesizer system, where there is a cache between the synthesizer and a larger, disk-based dictionary. A word is hashed using `gperf`, and if the word is already in the cache it is not looked up in the dictionary.

Since automatic static search set generators perform well in practice and are widely and freely available, there seems little incentive to code keyword recognition functions manually for most applications.

References

[1] M. Lesk and E. Schmidt, *LEX - A Lexical Analyzer Generator*. Bell Laboratories, Murray Hill, N.J., Unix Programmers Manual ed.

[2] S. Johnson, *YACC - Yet another Compiler Compiler*. Bell Laboratories, Murray Hill, N.J., Unix Programmers Manual ed.

[3] R. M. Stallman, *Using and Porting GNU CC*. Free Software Foundation, GCC 2.7.2 ed.

[4] A. Gokhale, D. C. Schmidt, and S. Moyer, "Tools for Automating the Migration from DCE to CORBA," in *Proceedings of ISS 97: World Telecommunications Congress*, (Toronto, Canada), IEEE Communications Society, September 1997.

[5] D. E. Knuth, *The Art of Computer Programming*, vol. 1: Searching and Sorting. Reading, MA: Addison Wesley, 1973.

[6] C. R. Cook and R. R. Oldehoeft, "A Letter Oriented Minimal Perfect Hashing Function," *SIGPLAN Notices*, vol. 17, pp. 18–27, Sept. 1982.

[7] A. Tharp and M. Brain, "Using Tries to Eliminate Pattern Collisions in Perfect Hashing," *IEEE Transactions on Knowledge and Data Engineering*, vol. 6, no. 2, pp. 329–347, 1994.

[8] R. J. Cichelli, "Minimal Perfect Hash Functions Made Simple," *Communications of the ACM*, vol. 21, no. 1, pp. 17–19, 1980.

[9] M. Brain and A. Tharp, "Near-perfect Hashing of Large Word Sets," *Software – Practice and Experience*, vol. 19, no. 10, pp. 967–978, 1989.

[10] R. Sprugnoli, "Perfect hashing functions: A single probe retrieving method for static sets," *Communications of the ACM*, pp. 841–850, Nov. 1977.

[11] G. V. Cormack, R. Horspool, and M. Kaiserwerth, "Practical Perfect Hashing," *Computer Journal*, vol. 28, pp. 54–58, Jan. 1985.

[12] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. M. auf der Heid, H. Rohnert, and R. Tarjan, "Dynamic Perfect Hashing: Upper and Lower Bounds," *SIAM Journal of Computing*, vol. 23, pp. 738–761, Aug. 1994.

[13] G. Jaeschke, "Reciprocal Hashing: A Method for Generating Minimal Perfect Hashing Functions," *Communications of the ACM*, vol. 24, pp. 829–833, Dec. 1981.

[14] T. Sager, "A Polynomial Time Generator for Minimal Perfect Hash Functions," *Communications of the ACM*, vol. 28, pp. 523–532, Dec. 1985.

[15] C. C. Chang, "A Scheme for Constructing Ordered Minimal Perfect Hashing Functions," *Information Sciences*, vol. 39, pp. 187–195, 1986.

[16] Bjarne Stroustrup, *The C++ Programming Language, 3rd Edition*. Addison-Wesley, 1991.

[17] D. C. Schmidt, "ACE: an Object-Oriented Framework for Developing Distributed Applications," in *Proceedings of the 6th USENIX C++ Technical Conference*, (Cambridge, Massachusetts), USENIX Association, April 1994.

[18] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

[19] D. Lea, "libg++, the GNU C++ Library," in *Proceedings of the 1st C++ Conference*, (Denver, CO), pp. 243–256, USENIX, Oct. 1988.

[20] J. Kegler, "A Polynomial Time Generator for Minimal Perfect Hash Functions," *Communications of the ACM*, vol. 29, no. 6, pp. 556–557, 1986.

[21] A. Gokhale and D. C. Schmidt, "Evaluating the Performance of Demultiplexing Strategies for Real-time CORBA," in *Proceedings of GLOBECOM '97*, (Phoenix, AZ), IEEE, November 1997.