GPERF

A Perfect Hash Function Generator

Technical Report 92-47

Douglas C. Schmidt and Tatsuya Suda
schmidt@ics.uci.edu and suda@ics.uci.edu
Department of Information and Computer Science,
University of California, Irvine,
Irvine, CA 92717, U.S.A.
(714) 856-4105 (phone)
(714) 856-4056 (fax) [1]

[1]This material is based upon work supported by the National Science Foundation under Grant No. NCR-8907909. This research is also in part supported by University of California MICRO program.

Abstract

gperf is a widely available perfect hash function generator written in C++. It automates a common system software operation: keyword recognition. **gperf** translates an n element user-specified keyword list *keyfile* into source code containing a k element lookup table and a pair of functions, `phash` and `in_word_set`. `phash` uniquely maps keywords in *keyfile* onto the range $0..k-1$, where $k \geq n$. If $k = n$, then `phash` is considered a *minimal* perfect hash function. `in_word_set` uses `phash` to determine whether a particular string of characters *str* occurs in the *keyfile*, using *at most* one string comparison.

This paper describes the user-interface, options, features, algorithm 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.

1 Introduction

Perfect hash functions are a time and space efficient implementation for static search sets, *e.g.*, compiler reserved words. Many articles describe perfect hashing[1, 2, 3, 4, 5] and minimal perfect hashing algorithms.[6, 7, 8, 9, 10, 11] However, few articles describe the design and implementation of a general-purpose perfect hashing generator tool in detail.[12] **gperf** is one such utility, it constructs perfect hash functions from a user-supplied keyword list.

gperf is intended as a practical "software-tool generating-tool," in the spirit of the UNIX utilities **lex**[13] and **yacc**.[14] The goal of **gperf** is to completely automate the perfect hash function generation process. **gperf** was originally developed to automate keyword recognizer construction for the GNU C and GNU C++ compilers. It removes the drudgery associated with constructing time and space efficient keyword recognizers by hand.

gperf is designed to run quickly for keyword sets up to approximately 1,000 keys. In addition, internal data structure and algorithms described below enable **gperf** to operate on keyword sets containing over 15,000 keywords. **gperf** generates efficient ANSI and K&R C, C++, or Ada source code as output.

gperf was inspired by Keith Bostic's utility "**perfect**" distributed to `net.sources` in 1984. C++ source code for **gperf** is available via anonymous ftp from **ics.uci.edu** (128.195.1.1) and is also distributed along with the GNU libg++ library. The distribution includes keyfiles for Ada, C, Pascal, C++, Modula 2, and Modula 3 reserved keywords. Finally, a highly portable, functionally equivalent K&R C version of **gperf** is archived in volume 20 of comp.sources.unix.

gperf's output is used as the reserved keyword recognizer in lexical analyzers for several production and research compilers and language processing tools, including GNU C, GNU C++, GNU Pascal, GNU Modula 3, and GNU **indent**.[15, 16] In addition to the GNU compilers, other known **gperf** applications include:

- 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 16 minutes of CPU time on a 16 MHz Sun 4/260.
- the GNU **indent** C code reformatting program, where the inclusion of perfect hashing sped up the program by an average of 10 percent.
- hash functions for assembly mnemonics in the 80x86, 680x0, Z8000, and MIPS RISC instruction sets.
- a public domain program converting double precision FORTRAN source code to/from single precision uses **gperf** to modify subroutine names that depend on the types of their arguments, *e.g.*, replacing `sgefa` with `dgefa` in the LINPACK benchmark. Each name corresponding to a subroutine 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.

The remainder of the paper is organized as follows: Section 2 describes various search structure implementations and compares them against **gperf**-generated hash tables; Section 3 presents a sample input keyfile; Section 4 discusses various design and implementation issues; and Section 5 shows the results from empirical benchmarks between **gperf**-generated recognizers and other popular techniques for reserved word lookup.

2 Static Search Structure Implementations

gperf generates source code that implements a static search structure. A *static search structure* is an abstract data type with certain fundamental operations, such as *initialize*, *insert*, and *retrieve*. It is a useful data structure for representing *static search sets*.[1] Static search sets are common in system software applications. Typical static search sets include compiler reserved words, assembler instruction mnemonics, and built-in shell interpreter commands. Search set elements are called *keywords*. Keywords are inserted into the structure once, usually at compile-time.

Numerous static search structure implementations exist, *e.g.*, 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.[17] Different approaches offer trade-offs between memory utilization and search time efficiency. For example, an n element sorted array is space efficient, though the average- and worst-case time complexity for retrieval operations using binary search on a sorted array is proportional to $O(\log n)$.[17] Conversely, chained hash table implementations locate a table entry in constant, *i.e.*, $O(1)$, time on the average. However, they typically impose additional memory overhead for link pointers and/or unused hash table buckets and also exhibit $O(n^2)$ worst-case performance.[17]

A minimal perfect hash function is a static search structure implementation defined by two properties:

1. **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.
2. **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.[17] However, several variations are also useful for many practical hashing applications, especially ones involving hundreds or 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 main reasons to generate non-minimal hash functions:
    1. Generating non-minimal perfect functions can be substantially faster than generating *minimal perfect* hash functions.[8, 2]
    2. Non-minimal perfect hash functions can also execute faster than minimal ones when searching for elements that are *not* in the table. This situation often occurs when recognizing reserved words in program source code.[6, 18]
- **Near-Perfect Hash Functions**: Near-perfect hash functions do not possess the perfect property, since they allow non-unique keyword hash values[2] (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.

gperf has command-line options that instruct it generate minimal perfect, non-minimal perfect, and near-perfect hash functions.

3 Interacting with gperf

gperf reads a keyword list and optional *associated attributes* from a *keyfile* or from the standard input. Keywords are specified as arbitrary character strings delimited by a user-specified field separator defaulting to `','` (*i.e.*, keywords can 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 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 structurally similar to the UNIX utilities **lex** and **yacc**, and uses the following format:

```
%{
#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
%%
#ifdef DEBUG
int main () {
  char buf[80];
  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

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

A pair of consecutive `%` symbols in the first column separate declarations from the list of keywords and their optional attributes. C, C++, or Ada 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 can be placed at the end of the declaration section, just before the `%%` separator. This feature enables typed attribute initialization. In Figure 1, for example, `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. Any additional attribute fields 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 '`\n`' by default). The 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 do not possess any 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," etc. from consideration in a "key-word-in-context" indexing application.[17]

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

4 Design and Implementation Issues

gperf is written in approximately 4,500 lines of C++ source code.[19, 20] C++ was chosen as the implementation language because it supports data abstraction and information hiding better than C, while still maintaining C's efficiency and expressiveness.[21] The following section explains **gperf**'s internal data structures, outlines its perfect hash function generation algorithm, examines its generated source code output, describes several reusable class components, and discusses the program's current limitations.

4.1 Internal Data Structures

gperf's implementation involves two important internal data structures: *keyword signatures* and the *associated values array*.

4.1.1 Keyword Signatures

Every user-specified keyword and its attributes are read from the keyfile and stored on a linked list node. **gperf** only considers a subset of each keywords' characters while searching for a perfect hash function solution. The subset is called the "keyword signature," or *keysig*; it contains the particular subset of characters used by the automatically generated recognition function to compute a keyword's hash value. Keysig's are also created and cached in each linked list node when the keyfile is initially processed.

Users can control the generated hash function's contents by explicitly specifying the keyword index positions to use as keysig elements. Choosing different key positions is easily expressed in **gperf**'s command-line syntax using the `'-k'` option. The default is `'-k 1,$'`, where the `'$'` represents the keyword's final character. However, a keysig is actually a *multiset* or *bag*, as it may contain multiple occurrences of certain characters. This approach differs from other perfect hash function methods, where only the keyword's first and last characters, plus its length, are examined when computing the hash value.[6]

The generated hash function properly handles keywords shorter than a specified index position by skipping characters that exceed the keyword's length. Users can also instruct **gperf** to include all of a keyword's characters in its keysig via the `'-k*'` option. Table 1 shows the keywords, keysigs, and hash value for each month shown in the Figure 1 keyfile.

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

4.1.2 Associated Values Array

The *associated values* array is a data structure closely related to keysigs; it is indexed by keysig characters. The array is constructed internally by **gperf**, referenced frequently during **gperf**'s execution, and later output in the generated hash function as a `static` local array. This array is declared as "`unsigned int asso_values[MAX_ASCII_SIZE]`." When searching for a perfect hash function solution, **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*.

By 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 computed by combining the associated values of its keysig with its length (the '`-n`' option instructs **gperf** not include the length of the keyword when computing the hash function). 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.[22] For example, using this default scheme for C++ causes a collision between the `delete` and `double` reserved words. Resolving this collision and generating a perfect hash function for C++ reserved words requires adding an additional character to the keysig via the '`-k`' option with parameters '`1,2,$`', *i.e.*:

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

4.2 Perfect Hash Function Generation

gperf's three main phases for generating a perfect or near-perfect hash function are:

1. Process command-line options, read keywords and attributes (the input format is described in Section 3), and initialize internal data structures (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 below).
3. Generate formatted C, C++, or Ada code according to the command-line options (output format is described in Section 4.3 below).

These next subsections gives a detailed description of **gperf**'s non-backtracking search algorithm used in the second phase mentioned above.

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 mapped keywords by 1. It succeeds if the hash value computed for keyword i does not collide with the previous $i - 1$ uniquely hashed keywords, *i.e.*:

Algorithm 1

```
for $i \leftarrow 1$ to $n$ loop
    if phash ($i^{th}$ key) collides with any phash ($1^{st}$ 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
```

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

4.2.2 Collision Resolution Strategies

Disjoint Union As outlined in Algorithm 1 above, **gperf** attempts to resolve keyword hash collisions by modifying certain associated values. To avoid performing unnecessary work, **gperf** is selective when changing associated values. 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 $\{A \cup B\} - \{A \cap B\}$. Note that no other associated values can possibly resolve the collision at this point.

For instance, the keywords `january` and `march` have the keysigs `'an'` and `'ar'`, respectively (see Table 1). A collision occurs during **gperf**'s execution when `asso_values['a']`, `asso_values['n']`, and `asso_values['r']` all equal 0 (note that since the `'-n'` option is used, the different keyword lengths are not considered in the resulting hash function). When **gperf** resolves this collision it only considers changing the associated values for `'n'` and/or `'r'`. Changing `'a'` by any increment will not resolve the collision, since `'a'` occurs the same number of times in each keysig.

By default, all `asso_values` are initialized to 0, and when a collision is detected **gperf** increments the selected associated value by 5. The command-line option `'-j'` can be used to increment by a random amount or by any fixed amount. In the months example, the `'-j 1'` option was used, so **gperf** quickly resolves the collision between `january` and `march` by incrementing `asso_value['n']` by 1 (which also turns out to be its final value, as shown in Table 1).

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

Heuristics As a heuristic, characters in the disjoint union are sorted by increasing frequency of occurrence, so that less frequently used characters are changed before more frequently used characters. The assumption here is that changing less frequently 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.

A perfect hash function is achieved if the systematic changes to the associated values configuration described in the previous paragraph eliminate all keyword collisions upon reaching the end of the keyword list. 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 rarely in practice.

Many perfect hash function generation algorithms are sensitive to the order that keywords are considered.[8, 2] If the `-o` command-line option is enabled, **gperf** mitigates this effect by optionally reordering the keywords before invoking the main algorithm. This reordering is done in a two stage pre-pass that applies two common heuristics described by Cichelli.[18] First, the keyword list is sorted by decreasing frequency of keysig characters' occurrence. The second reordering pass then places keys with "already determined keysig values" earlier in the keylist.

These two heuristics potentially prune the search space by handling inevitable collisions early in the generation process. If **gperf** can resolve these collisions quickly by changing the appropriate associated values it will run faster on many keyword sets and often decrease the perfect hash function range. On the other hand, if the number of keywords is large and the user wishes to generate a near-perfect hash function, this reordering sometimes *increases* **gperf**'s execution time, since collisions begin earlier and frequently persist throughout the remainder of keyword processing. Additional details and rationalizations for these reordering heuristics are discussed by Cichelli and Brain.[18, 2]

```
static unsigned int phash (const char *str, 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]];
}
```

Figure 2: The `phash` Function

4.3 Output Format

Figure 8 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 4/260, *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 output.

4.3.1 Generated Symbolic Constants

gperf's output contains seven symbolic constants that summarize the results of applying Algorithm 8 to the keyfile, *e.g.*:

```
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
};
```

A *minimal perfect* hash function occurs 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

When given a keyfile as input, **gperf** attempts to generate a perfect hash function that uses at most one string comparison to recognize keywords in the lookup table. **gperf** produces a lookup table called `asso_values`, shown in Figure 2. `asso_values` is used by the two generated functions that compute hash values and perform table lookup.

The lookup table is implemented by either an array or a `switch` statement (note, the generated Ada code uses a `case` statement rather than a `switch` statement). An array is generated by default, emphasizing run-time speed over minimal memory utilization. However, there are command-line options that allow trading-off memory for execution-time. For example, expanding the range of hash values produces a sparser lookup table. This generally yields faster keyword searches but requires additional memory.

The array-based method 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 shown in Figure 3 can become extremely large. Several problems arise in this case, (1) the time to compile the sparsely populated array is excessive, (2) the array size may be too large to store in main memory, or (3) a large array may lead to increased thrashing in virtual memory environments.

```
const struct months *is_month (const char *str, 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 = phash (str, len);

      if (key <= MAX_HASH_VALUE && key >= MIN_HASH_VALUE)
        /* ... see text ... */
  }
  return 0;
}
```

Figure 3: The `is_month` Function

To handle the problems mentioned 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. Note that more than one `switch` statement may be required, since many C compilers do not generate correct code for extremely large `switch` statements *e.g.*, greater than 10,000 cases. Figure 4 shows how the `switch` statement code appears if the months example is generated with **gperf**'s `'-S 1'` option.

Since the months example is somewhat contrived, the trade-off between the array and `switch` approach is not particularly obvious. However, a good compiler can 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.[16] 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, since 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 `phash` and `in_word_set`, although a different name can 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, `char *str`, and a length parameter, `int len`.

The Generated Hash Function (`phash`) Figure 2 shows the `phash` function generated from the input keyfile shown in Figure 1. Since the command-line option `'-k 2, 3'` was enabled, `phash` returns an `unsigned int` value calculated by indexing the keysig characters (in this case ASCII values of the second and third characters) from its `str` argument into the local `static` array `asso_values` (C arrays start at 0, so `str[1]` is actually the second character). The two resulting numbers are added together to compute `str`'s hash value. The `asso_values` array is constructed by **gperf**; it maps the user-defined keywords onto

```
{
  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 4: The `switch`-based Lookup Table

unique hash values (additional details are described in Section 4.1.2).

Note that 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 2) represent ASCII characters that do not occur as either the second or third characters in the months of the year. This information is used by the `is_month` function shown in Figure 3 to quickly eliminate input strings that cannot possibly be month names.

Generated Lookup Function (`in_word_set`) The `in_word_set` function is the interface to the perfect hash lookup routines (the `phash` function is declared `static` and is not directly invoked by application programs). If the function's first parameter, `char *str`, is a valid user-define keyword then `in_word_set` returns a pointer to the corresponding record containing each keyword and its associated attributes, otherwise a NULL pointer is returned.

Figure 3 shows the `in_word_set` function, renamed to `is_month` for the current example via the '`-N`' command-line option. Note how **gperf** checks the `len` parameter and resulting `phash` function return value against the symbolic constants for `MAX_WORD_LENGTH`, `MIN_WORD_LENGTH`, `MAX_HASH_VALUE`, and `MIN_HASH_VALUE`. This quickly eliminates many non-month names from further consideration. If users know in advance that all input strings are valid keywords, **gperf** can be instructed to 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 (filling in the `/* ... see text ... */` comment from Figure 3):

```
{
  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 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 (note that 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 routine when the `str`'s first letter does not match any of the keywords in the lookup table).

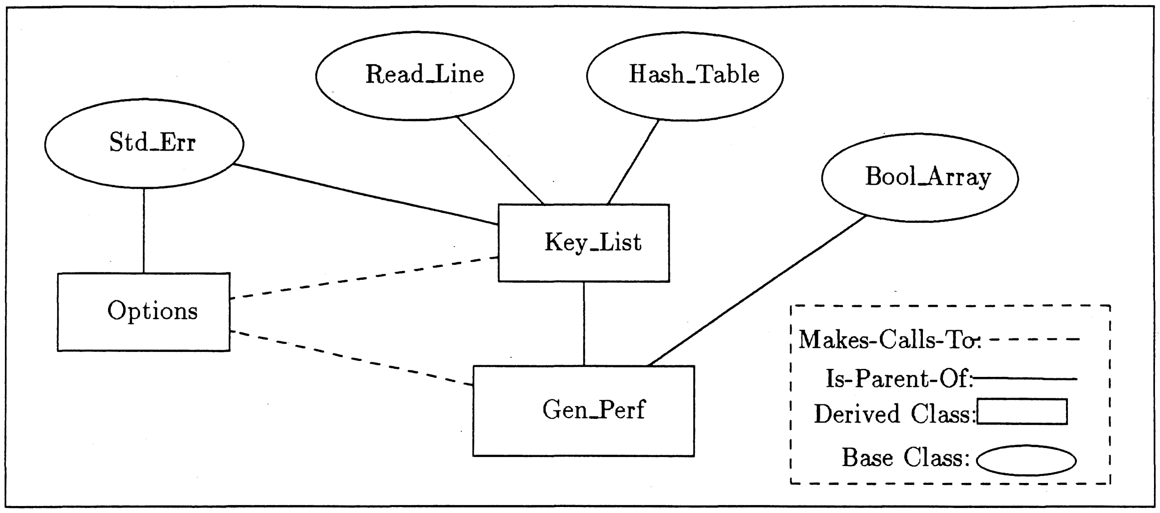

Figure 5: **gperf**'s Inheritance Hierarchy

4.4 Reusable Class Components

Figure 5 illustrates **gperf**'s overall program structure. **gperf** is constructed from reusable components that also serve as base-classes in a "forest"-style library.[23] Each of these classes evolved "bottom-up" from special-purpose utilities into reusable software components. Several noteworthy reusable classes include the following abstract data types:

- **Std_Err:** This class standardizes and consolidates the formatting of error messages throughout **gperf**. `Std_Err` generalizes the functionality of the UNIX `perror` library routine. A `static` class member function, with type signature `void Std_Err::report_error(const char *, ...)`, interprets a `printf`-style format string containing directives that provide a uniform error handling facility. Standard services include: (1) writing a user specified error message to the standard error stream, (2) formatting and printing various common data types passed as arguments to the variadic function, (3) displaying appropriate system error messages corresponding to failed system and library calls, (4) aborting the program with a specified exit code, (5) calling function-pointers passed as parameters, and (6) displaying the name used to invoke the main program in error messages.
- **Read_Line:** Each line in **gperf**'s input contains a single keyword followed by any optional associated attributes, ending with a newline character ('`\n`'). The `'Read_Line::get_line'` member function copies an arbitrarily long '`\n`'-terminated string of characters from the input into a dynamically allocated buffer. A recursive auxiliary function, `'Read_Line::readln_aux'`, insures only one call is made to the free store allocator per input line read, *i.e.*, there is no need for reallocating and resizing buffers dynamically. This routine proved useful enough to be incorporated as an extension in the GNU libg++ **stream** library.[24]
- **Hash_Table:** This class provides a search structure implemented via double hashing.[17] 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!

- **Bool_Array:** Earlier versions of **gperf** were instrumented with a run-time code profiler. The results showed that **gperf** spent approximately 90 to 99 percent of its time in a single routine when performing Algorithm 1 (described in Section 4.2.1) on large input keyfiles that evoke many collisions. This one routine, `'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 routine is called so frequently, it is important that it exhibits minimal execution overhead. **gperf** employs a novel boolean array abstract data type called `Bool_Array` to expedite this process. The C++ interface for the `Bool_Array` class is depicted in Figure 6. All class data and member functions are declared with storage class `static`, since only one copy of `Bool_Array` is required (this reduces run-time overhead since no "`this`" pointer is passed during function calls).

  Class member function `Bool_Array::in_set(int)` efficiently detects duplicate keyword hash values for a given associated values configuration, returning non-zero if a value is already in the set and zero otherwise. Whenever a duplicate is detected, `Bool_Array::reset()` is called to reset all the array elements back to "empty" for ensuing iterations of the search process.

  If many hash collisions occur, `Bool_Array::reset()` is executed frequently during the duplicate detection and elimination process. 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 `Bool_Array::array` back to empty, especially when the number of keywords actually checked for duplicate hash values is comparatively small. Algorithm 2 describes a technique called *generation numbering* that is used to optimize this process by not explicitly reinitializing the entire array.

  Algorithm 2 The generation numbering technique operates as follows:

  1. *The class constructor dynamically allocates space for k* `unsigned short` *integers and points* `Bool_Array::array` *at the allocated memory. All k array elements in* `Bool_Array::array` *are initially assigned 0 (representing "empty") and the* `Bool_Array::generation_number` *counter is set to 1.*
  2. *The* `Bool_Array::in_set(int)` *member function is used to detect duplicate keyword hash values. If the number stored at the* `phash(keyword)` *index position in* `Bool_Array::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 immediately assigned to the* `phash(keyword)` *array location, thereby marking it as a duplicate if it is subsequently referenced during this particular iteration of the search process.*
  3. *Otherwise, if the value at location* `Bool_Array::array [phash(keyword)]` *is equal to the generation number, a duplicate exists and the algorithm must try modifying certain associated values to resolve the collision.*
  4. *If a duplicate is detected, the* `Bool_Array::array` *elements are reset to empty for subsequent iterations of the search process.* `Bool_Array::reset()` *simply increments the value of* `Bool_Array::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 (this occurs infrequently in practice).*

A design principle employed throughout **gperf**'s implementation is "first determine a clean 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

```
class Bool_Array
{
private:
  typedef unsigned short TYPE;     // Using unsigned short saves space
  static TYPE generation_number;   // Current generation count
  static TYPE *array;              // Dynamically allocated storage buffer
  static int size;                 // Length of dynamically allocated array

public:
  Bool_Array (int k);              // Allocate a k element dynamic array
 ~Bool_Array (void);               // Returns dynamic memory to free store
  static int in_set (int value);   // Checks if 'value' is a duplicate
  static void reset (void);        // Reinitializes all set elements to FALSE
};
```

Figure 6: Boolean Array Abstract Data Type

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.

4.5 Current Compromises and Limitations

Several other hash function generation algorithms utilize some form of backtracking when searching for a perfect or minimal perfect solution.[8, 6, 2] For example, Cichelli's 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, since even "intelligently-guided" exhaustive search quickly becomes impractical for more than several hundred keywords.

To simplify Algorithm 1 and improve average-case performance, **gperf** does not backtrack when keyword hash collisions occur. **gperf** may process the entire keyfile input, therefore, *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 either 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 can handle keyword membership queries efficiently, since only a small number of duplicates usually remain (the exact number depend on the keyword set and the command-line options).

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 7 shows the relevant fragment of the generated near-perfect hash function code.

```
{
  char *rw;
  ...
  switch (phash (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 7: The Near-Perfect Lookup Table Fragment

Executable	**Input File**					
Program	`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

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 can edit the generated code by hand or via an automated script to completely disambiguate the search key.

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-quality applications unless the resulting executable code speed is competitive with typical

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

alternative implementations. In fact, it has been argued that there are *no* circumstances where perfect hashing proves worthwhile, compared with other common static search structure methods.[25]

To compare the efficacy of the **gperf**-generated perfect hash functions against other common static search structure implementations, seven test programs were developed and executed on six large input files. Each test program implemented the same function: a recognizer for the 71 GNU G++ reserved words. 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; these statistics are based upon the `malloc` routine from the GNU libg++ 1.37 library).

- **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 routine similar to the one that recognizes reserved words for AT&T's **cfront** 2.0 C++ compiler. The table's load factor is 0.39, the same as it is in **cfront** 2.0, *i.e.* $\frac{71}{181}$ for **chash.exe** versus $\frac{48}{123}$ for **cfront** 2.0.
- **patricia.exe:** a PATRICIA trie recognizer, where PATRICIA stands for "Practical Algorithm to Retrieve Information Coded in Alphanumeric."[26] A complete PATRICA trie implementation is available in the GNU libg++ class library distribution.
- **binary.exe:** a carefully coded binary search routine 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 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).

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

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) {
  const int MAX_ID = 80; /* Larger than any input identifier. */
  char buf[MAX_ID];
  while (gets (buf))
    printf ("%s", buf);
}
```

All of the above reserved word recognizer programs were compiled by the GNU G++ 1.37 compiler with the `'-O -fstrength-reduce -finline-functions -fdelayed-branch'` options enabled. They were then tested on an otherwise idle 16 MHz Sun 4/260 with 32 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. These systems included the ET++ windowing toolkit (`ET++.in`), the NIH class library (`NIH.in`), the GNU G++ 1.37 C++ compiler (`g++.in`), the **idraw** figure drawing utility from the InterViews 2.6 distribution (`idraw.in`), the AT&T **cfront** 2.0 C++ compiler (`cfront.in`), and the GNU libg++ 1.37 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 structure 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:

- 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.
- 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 may 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** can generate recognizers for the 15,400 line MEDLINE keyfile input, because the size of the state machine is too large for their internal DFA state tables.

6 Future Directions and Conclusions

Fully automating the perfect hash function generation process remains **gperf**'s most significant unfinished extension. One approach is to replace **gperf**'s current algorithm with a more exhaustive approach, *e.g.*, Brain and Tharp's enhancements to Cichelli's algorithm.[2] Due to **gperf**'s object-oriented program design, these modifications will not unduly disrupt the overall program structure. The perfect hash function generation module, `class Gen_Perf`, is essentially 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 routines can be extended to generate code for other languages, *e.g.*, a Modula 2/Module 3 module or an Eiffel class.

gperf's was originally designed to automate compiler keyword recognizer construction. The various features described in this paper enable it to achieve its goal, as evidenced by its use in the GNU compiler and language processing tools. **gperf** also works well on larger keyword sets, as evidenced by the 15,400 line MEDLINE data. Since automatic static search structure generators perform well in practice and are widely and freely available, there seems little incentive to code keyword recognition functions by hand for most applications.

Acknowledgments

In addition to Keith Bostic, who initially inspired **gperf**, special thanks is extended to Michael Tiemann and Doug Lea. Michael wrote the GNU G++ compiler. Doug gave me a forum in GNU libg++ to exhibit my creation; he also commented on drafts of this paper. Adam de Boor and Nels Olson contributed many insights that greatly helped improve the quality and functionality of **gperf**. Vern Paxson provided an efficient **flex** input specification file for the GNU C++ keywords. Finally, Rolf Ebert extended **gperf** to generate Ada code.

```c
/* C code produced by gperf version 2.5 (GNU C++ version) */
/* Command-line: gperf -C -p -a -n -t -o -j 1 -k 2,3 -N is_month months.gperf  */
#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 phash (const char *str, 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, 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 = phash (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;
}
#ifdef DEBUG
int main () {
  char buf[80];
  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 8: Minimal Perfect Hash Function Generated by **gperf**

References

[1] R. Sprugnoli. Perfect Hashing Functions: A Single Probe Retrieving Method for Static Sets. *Communications of the ACM*, 11:841–850, November 1977.

[2] M. D. Brain and A. L. Tharp. Near-Perfect hashing of large word sets. *Software - Practice and Experience*, 19:967–978, 1989.

[3] C.C. Chang and T. Wu. A Letter-oriented Perfect Hashing Scheme Based upon Sparse Table Compression. *Software Practice and Experience*, 21:35–49, 1991.

[4] G.V. Cormack, R.N.S. Horspool, and M. Kaiserwerth. Practical Perfect Hashing. *Computer Journal*, 28:54–58, January 1985.

[5] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, and R.E. Tarjan. Dynamic Perfect Hashing: Upper and Lower Bounds. In 29^{th} *Focus on Computer Science*, pages 524–531.

[6] Richard J. Cichelli. Author's Response to "On Cichelli's Minimal Perfect Hash Functions Method". *Communications of the ACM*, 23:729, December 1980.

[7] G. Jaeschke. Reciprocal Hashing: A Method for Generating Minimal Perfect Hashing Functions. *Communications of the ACM*, 24:829–833, December 1981.

[8] C.R. Cook and R.R. Oldehoeft. A Letter Oriented Minimal Perfect Hashing Function. *SIGPLAN Notices*, 17:18–27, September 1982.

[9] Thomas J. Sager. A Polynomial Time Generator for Minimal Perfect Hash Functions. *Communications of the ACM*, 28:523–532, December 1986.

[10] E. A. Fox, Q. F. Shen, L. S. Heath, and S. Datta. A more cost effective algorithm for finding minimal perfect hashing functions. *ACM Conference Proceedings*, pages 114–122, 1989.

[11] C.C. Chang. A Scheme for Constructing Ordered Minimal Perfect Hashing Functions. *Information Sciences*, 39:187–195, 1986.

[12] Douglas C. Schmidt. GPERF: A Perfect Hash Function Generator. In *USENIX C++ Conference Proceedings*, pages 87–102. USENIX Association, April 1990.

[13] M. Lest and E. Schmidt. Lex - A Lexical Analyzer Generator. *UNIX Programmer's Manual: Supplementary Documents 1.*, 1986.

[14] Steven Johnson. Yacc: Yet Another Compiler Compiler. *UNIX Programmer's Manual: Supplementary Documents 1.*, 1986.

[15] Michael D. Tiemann. User's Guide to GNU C++. *Free Software Foundation*, 1991.

[16] Richard M. Stallman. Using and Porting GNU CC. *Free Software Foundation*, 1991.

[17] Donald Knuth. *The Art of Computer Programming, VOL 3: Sorting and Searching*. Addison-Wesley, 1973.

[18] Richard J. Cichelli. Minimal Perfect Hash Functions Made Simple. *Communications of the ACM*, 23:17–19, January 1980.

[19] Bjarne Stroustrup and Margret Ellis. *The Annotated C++ Reference Manual*. Addison-Wesley, 1986.

[20] Bjarne Stroustrup. *The C++ Programming Language (Second Edition)*. Addison-Wesley, 1986.

[21] Bjarne Stroustrup. What is Object-Oriented Programming? *IEEE Software*, pages 10–20, May 1988.

[22] G. Jaeschke and G. Osterburg. On Cichelli's Minimal Perfect Hash Functions Method. *Communications of the ACM*, 22:728–729, December 1980.

[23] Doug Lea. libg++, The GNU C++ Library. In *USENIX C++ Conference Proceedings*, pages 243–256. USENIX Association, October 1988.

[24] Doug Lea. User's Guide to GNU C++ Class Library. *Free Software Foundation*, 1991.

[25] Jeffrey Kegler. A Polynomial Time Generator for Minimal Perfect Hash Functions. *Communications of the ACM*, 29:556–557, June 1986.

[26] Robert Sedgewick. *Algorithms*. Addison-Wesley, 1983.

[27] M.L. Fredman, J. Komlos, and E. Szemeredi. Storing a Sparse Table with $O(1)$ Worst Case Access Time. *Journal of the ACM*, pages 538–544, July 1984.

[28] R.W. Sebesta and M.A. Taylor. Minimal Perfect Hash Functions for Reserved Word Lists. *SIGPLAN Notices*, 20:47–53, September 1985.

[29] C.C. Chang. The Study of an Ordered Minimal Perfect Hashing Scheme. *CACM*, 27:384–387, 1984.