A Dictionary Implementation Based on Dynamic Perfect Hashing

MARTIN DIETZFELBINGER
Technische Universität Ilmenau
MARTIN HÜHNE
Fachhochschule Südwestfalen
and
CHRISTOPH WEIDLING
DocuWare AG

We describe experimental results on an implementation of a dynamic dictionary. The basis of our implementation is "dynamic perfect hashing" as described by Dietzfelbinger et al. (*SIAM J. Computing 23*, 1994, pp. 738–761), an extension of the storage scheme proposed by Fredman et al. (*J. ACM* 31, 1984, pp. 538–544). At the top level, a hash function is used to partition the keys to be stored into several sets. On the second level, there is a perfect hash function for each of these sets. This technique guarantees $O(1)$ worst-case time for lookup and expected $O(1)$ amortized time for insertion and deletion, while only linear space is required. We study the practical performance of dynamic perfect hashing and describe improvements of the basic scheme. The focus is on the choice of the hash function (both for integer and string keys), on the efficiency of rehashing, on the handling of small buckets, and on the space requirements of the implementation.

Categories and Subject Descriptors: E.2 [**Data**]: Data Storage Representation; F.2.2 [**Theory of Computation**]: Nonnumerical Algorithms and Problems

General Terms: Algorithms, Experimentation, Performance

Additional Key Words and Phrases: Data structures, dictionaries, dynamic hashing, hash functions, implementation

This work was supported in part by DFG grant Di 412/5-1. Part of this work was done while the first and the second author were affiliated with Universität Dortmund; the third author was affiliated with Technische Universität Ilmenau while this work was done.

Authors' addresses: Martin Dietzfelbinger, Technische Universität Ilmenau, Fakultät für Informatik und Automatisierung, Postfach 100565, 98684 Ilmenau, Germany; email: martin.dietzfelbinger@tu-ilmenau.de. Martin Hühne, Fachhochschule Südwestfalen, Fachbereich Informatik und Naturwissenschaften, Postfach 2061, 58590 Iserlohn, Germany; email: hashing@huehne.fh-swf.de. Christoph Weidling, DocuWare AG, Therese-Giehse-Platz 2, 82110 Germering, Germany; email: christoph.weidling@gmx.de.

ACM Reference Format:
Dietzfelbinger, M., Hühne, M., and Weidling, C. 2008. A dictionary implementation based on dynamic perfect hashing. ACM J. Exp. Algor. 12, Article 1.11 (June 2008), 25 pages DOI = 10.1145/1370596.1370602 http://doi.acm.org 10.1145/1370596.1370602

1. INTRODUCTION

A dictionary is a fundamental data structure. The basic objects for a dictionary are *keys* (the set of keys is denoted by U) and data items (the set of data items is denoted by R). A dictionary stores a finite set of keys, where each key is associated with a data item. Formally, a dictionary is a mapping $f: S \to R$ for a finite subset S of U. The operations of a dynamic dictionary are:

—construct (creates an empty dictionary);
—insert(x, r) (replaces f by $f' \cup \{(x, r)\}$, where $f' = f - \{(x, f(x))\}$, if x is in the domain of f and $f' = f$ otherwise);
—delete(x) (replaces f by $f - \{(x, f(x))\}$, if x is in the domain of f);
—lookup(x) (returns $f(x)$ if x is in the domain of f and "not found" otherwise);
—destroy (destroys the dictionary and frees the storage).

Many different implementations for dictionaries are known. Most of them are based on hashing or on search trees. For information on these standard methods and their analysis, see any textbook on algorithms and data structures.

In this study, we consider practical experiments with an implementation of a dictionary on the basis of *dynamic perfect hashing* [Dietzfelbinger et al. 1994]. This method has one central property: guaranteed constant lookup time in the worst case, asymptotically, i. e., for very large sets S. For the analysis to work, it is required that universal classes of hash functions [Carter and Wegman 1979] are employed. This notion is described in Section 2. Usually, for key sets of everyday size (up to a few million keys), standard implementations of dictionaries will be faster and will use less space than implementations based on dynamic perfect hashing. It is not the purpose of this study to compare dynamic perfect hashing with other implementations of dictionaries, but to investigate certain intrinsic properties of variants of this special implementation.

The performance of the dictionary strongly depends on the time needed for evaluating the hash functions. Thus, to obtain information on what a good choice is for a class of hash functions, a preliminary study on the performance of such functions is included in the paper. Depending on the architecture, either a class based on simple arithmetic operations [Dietzfelbinger et al. 1997] or a class based on table lookups [Czech et al. 1992] turns out to be well suited. For the dictionary implementation, the latter class was used. Storing such a function requires a nonnegligible amount of memory space and initialization time. This has the effect that our dictionary implementation is not suitable for small sets of keys. As for the data structure proper, some modifications were made in comparison to the "pure" algorithm of Dietzfelbinger et al. [1994]. In the data structure, many small sets of keys are generated that are stored independently, using a multitude of subdictionaries. We save space by using the same hash

functions for many of these subdictionaries and save time by realizing very small subdictionaries by linked lists instead of hash tables. As predicted by the theory, for data sets that are large enough, we observe constant access times both for keys that are integers and for keys that are strings. The properties of the implementation vary as certain parameters of the implementation are changed.

The organization of this paper is as follows. The storage scheme and the concept of universal hashing is described in Section 2. In Section 3, the performance of some universal classes of hash functions is examined. In Section 4, some details of our implementation are described. The results of the experiments are contained in Section 5.

2. THE STORAGE SCHEME

Dynamic perfect hashing is an extension of the static storage scheme proposed by Fredman et al. [1984] (the "FKS scheme"). In this section, we outline the scheme and sketch one basic tool, the concept of a universal class of hash functions. For a detailed description and an analysis of the time and space complexity of these schemes, we refer to Dietzfelbinger et al. [1994] and Fredman et al. [1984].

We start with a description of the FKS scheme. Let $S \subseteq U$ be any fixed set of n distinct keys taken from a universe U. With each key in S, a piece of information is associated. The task is to find a linear space storage scheme such that for any $x \in U$ the lookup operation lookup(x) can be performed in constant time, i.e., we want to know whether the key x is in S and, if so, which information is associated with this key. The FKS scheme consists of two levels. At the top level, there is a *main table* with entries numbered $0, \ldots, m-1$ and a hash function h (the *level-1 function*) that maps the keys in S into this table. The keys $x \in S$ mapped to entry $h(x) = i$ of the main table form the *bucket* i. On the second level, for each bucket i, $i = 0, \ldots, m-1$, there is a *level-2 hash function* h_i, which maps the keys contained in this bucket into a *subtable* such that there are no collisions. Since the level-2 functions are perfect hash functions, lookup operations can be performed in constant time. Fredman et al. show how to construct the level-1 and level-2 hash functions and how to choose the size of the subtables such that the storage scheme uses only linear storage space (in the number of items stored).

In addition to lookup operations, dynamic dictionaries have to cope with update operations (insert, delete). Inserting new keys may cause collisions with keys already stored in the dictionary. Deleting keys may cause too much space to be used for the remaining keys. In this case, it is necessary to *rehash*, i.e., to reorganize subtables or to reorganize the whole table "on the fly."

The basis of both the FKS scheme and dynamic perfect hashing is the concept of universal hashing. (Although this is not made explicit in Fredman et al. [1984], their construction uses a universal class.) This concept is because of Carter and Wegman [1979]. A class $\mathcal{H}_m$ of functions from U to $\{0, \ldots, m-1\}$ is called *c-universal*, $c > 0$, if for all different keys $x, y \in U$ we have

$$\Pr_{h \in \mathcal{H}_m}(h(x) = h(y)) \leq \frac{c}{m}.$$

(In Section 3, some well-known universal classes of hash functions are reviewed.)

For a hash function $h \in \mathcal{H}_m$ and a bucket $i \in \{0, \ldots, m-1\}$, let $b_{h,i}$ denote the size $|\{x \in S : h(x) = i\}|$ of bucket i. Observe that for any key set S, the expected number $\mathrm{E}(|\{\{x, y\} \subseteq S : h(x) = h(y)\}|)$ of collisions is, at most, $\binom{n}{2}\frac{c}{m}$, if h is uniformly drawn from $\mathcal{H}_m$. Since the number of collisions w.r.t. hash function h is just $\sum_{i=0}^{m-1}\binom{b_{h,i}}{2}$, this yields ([Carter and Wegman 1979; Fredman et al. 1984])

$$\mathrm{E}_{h\in\mathcal{H}_m}\left(\sum_{i=0}^{m-1} {b_{h,i}}^2\right) = \mathrm{E}_{h\in\mathcal{H}_m}\left(\sum_{i=0}^{m-1} b_{h,i} + 2\binom{b_{h,i}}{2}\right) \leq n + \frac{cn(n-1)}{m}. \qquad (1)$$

Obviously, $\mathrm{E}_{h\in\mathcal{H}_m}(\sum_{i=0}^{m-1} {b_{h,i}}^2)$ is at least n. Thus, for $m \geq 2cn(n-1)$, at least one-half of the functions in $\mathcal{H}_m$ are perfect hash functions for S. Moreover, (1) yields that $\mathrm{E}_{h\in\mathcal{H}_m}(\sum_{i=0}^{m-1} {b_{h,i}}^2) = O(n)$, if $m = \Omega(cn)$.

In order to construct a perfect hashing scheme, the size of the tables is appropriately fixed. (For brevity, we omit constant factors in the following. Nonetheless it should be clear that the parameter c influences the space complexity of the storage scheme and the time complexity of updates (cf. Dietzfelbinger et al. [1994].) First, the size $m = \Theta(n)$ of the main table is fixed. Hence, for a constant fraction of functions in a universal class $\mathcal{H}_m$, we have $\sum_{i=0}^{m-1} {b_{h,i}}^2 = O(n)$. Such a function can be found in expected linear time. Then, for each bucket i, a subtable of size $m_i = \Theta({b_{h,i}}^2)$ is allocated. If the subtable is large enough, a constant fraction of the level-2 hash functions in a universal class $\mathcal{H}_{m_i}$ perfectly maps the keys in the bucket into this subtable. Thus, for each bucket, a perfect hash function can be found in expected time $O(m_i)$. Moreover, the total space allocated for all tables is $O(n)$.

Neither the size of the main table nor the sizes of the subtables are known in advance. This problem is handled using the standard "doubling technique": whenever one of the subtables overflows, it is replaced by a new one whose capacity is larger by a constant factor. The main table and the subtables are chosen so large that during the next $\Theta(n)$ update operations, the expected number of rehashes for the main table and for each of the subtables is a constant. After a series of $\Theta(n)$ update operations, the size of the main table is adjusted, and the whole data structure is built anew. In this way, dynamic perfect hashing guarantees constant worst-case time for lookup and expected constant amortized time for insertion and deletion.

3. EVALUATION TIME OF HASH FUNCTIONS

In this section we are concerned with a question from the "challenge project ideas" [McGeoch 1996a]: How much computation time is required to hash and how can the hashing schemes be adapted to large keys and realistic key properties?

Several c-universal classes have been proposed in the literature. In principle, any of these classes can be used in the FKS scheme or in the dynamic perfect hashing scheme. However, the evaluation time is different for functions from different classes. In order to pick an appropriate class for our implementation,

Table I. A Description of the Machines Used in Our Experiments[a]

	Sparc 10	Sparc 5	Ultra 1	i568
Vendor	Sun	Sun	Sun	iX-Terminal
Processor type	SuperSPARC	μSparcII	UltraSPARC-1	Intel Pentium
Processor speed	33 MHz	70 MHz	167 MHz	133 MHz
Main memory	160 MB	80 MB	64 MB	64 MB (60 ns)
Virtual memory	339 MB	512 MB	355 MB	None
Instruction cache	20 KB	16 KB	16 KB	8 KB
Data cache	16 KB	8 KB	16 KB	8 KB
Secondary cache	None	None	512 KB	256 KB
Operating system	SunOS 4.1.4	SunOS 4.1.4	Solaris 2.5.5	Linux 1.2.12

[a]Note that machine "i568" is diskless and has a slow ethernet connection. Thus, it was used only for testing the integer hash functions in Section 3.2. In our department, it was used as a Linux-based X-terminal, as described in the German "iX" magazine.

Table II. Description of the Machines Used in Our Experiments for Changing the Resizing and Load Factor[a]

	AMD Athlon	Ultra IIi	Pentium III
Vendor	AMD	Sun	Intel
Processor type	Athlon K7 (Thunderbird)	UltraSPARC IIi	Intel Pentium III
Processor speed	1200 MHz	800 MHz	850 MHz
Main memory	1024 MB	384 MB	256 MB
Instruction cache	64 KB	16 KB	16 KB
Data cache	64 KB	16 KB	16 KB
Secondary cache	512 KB	1024 KB	256 KB
Operating system	Linux 2.4.7	Solaris 7	Linux 2.4.7

[a]See Sections 6.2 and 6.3.

we compare the performance of some c-universal classes. Of course, efficient hash functions are also of interest for other applications of hashing.

After this work was carried out, Thorup [2000] suggested new methods and tricks for the fast implementation of hash functions from universal classes. Thorup starts with a class similar to that described in Dietzfelbinger et al. [1997] ("linear class without primes") and utilizes the speed of the floating-point coprocessor in implementing the functions. Thorup's method for function evaluation could be plugged into our implementation of dynamic perfect hashing (for integer keys).

3.1 Hardware

Most of the following experiments were carried out on SUN workstations (Sparc 10, Sparc 5, and Ultra 1) and on a Pentium-based machine (i568) (see Table I). For some lengthy experiments of Sections 6.2 and 6.3, we used more advanced machines (see Table II).

3.2 Integer Hashing

We start with the case where the keys are integers that fit in one machine word, i.e., the universe is $U = \{0, \ldots, p-1\}$ for some number $p \leq 2^{32}$. Let m denote the size of the table.

The following classes from Carter and Wegman [1979] require that the size p of the universe is a prime.

—Multiplicative class: The multiplicative class $\mathcal{H}^0_{\text{lin}}$ consists of the $p-1$ hash functions $h_a(x) = ((ax) \bmod p) \bmod m$, $0 < a < p$. This class is 2-universal. (It was used in Fredman et al. [1984].)

—Linear class: The class $\mathcal{H}_{\text{lin}}$ consists of the $p(p-1)$ hash functions $h_{a,b}(x) = ((ax+b) \bmod p) \bmod m$, $0 < a < p$, $0 \le b < p$. The linear class is 1-universal.

—Polynomial classes: For $s \ge 2$, let $\mathcal{H}_{\text{poly}(s)}$ be the class of all polynomials $h_{(a_0,\ldots,a_{s-1})} = ((\sum_{i=0}^{s-1} a_i x^i) \bmod p) \bmod m$, $0 \le a_0, \ldots, a_{s-1} < p$. These classes are $(1 + m/p)$-universal.

Note that for appropriate values of p, e.g., p approximately a power of 2, the mod p operation can be performed without division (see Carter and Wegman [1979]).

In the following, we review classes where the size of the universe and the size of the table are powers of 2 ($p = 2^\ell$, $m = 2^k$). One arithmetic operation and two bit-shift operations are sufficient to evaluate the functions in the following class:

—Multiplicative class without primes [Dietzfelbinger et al. 1997]: Let $\mathcal{H}_{\text{mult}}$ consist of the $2^{\ell-1}$ functions $h_a(x) = (ax \bmod 2^\ell) \operatorname{div} 2^{\ell-k}$, $a \in U$ odd. This class is 2-universal.

We consider another way to construct efficient universal classes of hash functions. Of course, the class $\mathcal{H}_{\text{table}}$ of all functions $h : U \to \{0, \ldots, m-1\}$ is 1-universal. Yet, if the universe is large, it is of little practical value. To represent a function $h \in \mathcal{H}_{\text{table}}$, the values $h(0), \ldots, h(p-1)$ are stored in a table. Thus, evaluating h becomes very fast—it involves only one table lookup. However, the space requirements seem forbiddingly large.

In order to obtain efficient classes that can be stored in main memory, let t be an integer that divides ℓ. The hash function h is described by t tables $T_0, \ldots, T_{t-1}$ of size $2^{\ell/t}$ each. The entries in these tables are k-bit integers. Let $x_{t-1} \ldots x_0$ be the base $2^{\ell/t}$ representation of the key x to be hashed. The hash value $h(x)$ is then the exclusive-or of the t table entries $T_0(x_0), \ldots, T_{t-1}(x_{t-1})$. Let $\mathcal{H}_{t \text{ tables}}$ denote the class of all these functions. It is well known and quite obvious that this class is 1-universal (see e.g. Czech et al. [1992]). The evaluation of functions in $\mathcal{H}_{t \text{ tables}}$ involves only a few bit-operations and t table lookups.

3.3 Experiments with Integer Hash Functions

We carried out a series of experiments to estimate the time needed to evaluate the hash functions described in Section 3.2. We consider hash functions with range size $m = 2^{24}$. The size of U is $p = 2^{32} - 5$ for the multiplicative class $\mathcal{H}^0_{\text{lin}}$, for the linear class $\mathcal{H}_{\text{lin}}$, and for the polynomial class $\mathcal{H}_{\text{poly}(3)}$, whereas the size is $p = 2^{32}$ for the multiplicative class without primes $\mathcal{H}_{\text{mult}}$ and for the table classes $\mathcal{H}_{t \text{ tables}}$, $t = 2, \ldots, 10, 16$.

The multiplications and additions in the class $\mathcal{H}^0_{\text{lin}}$, $\mathcal{H}_{\text{lin}}$, and $\mathcal{H}_{\text{poly}(3)}$ are 64-bit operations. For the classes $\mathcal{H}^0_{\text{lin}}$ and $\mathcal{H}_{\text{lin}}$, two variants are considered,

Table III. Results of the First Experiment[a]

	Sparc 5	Ultra 1	i586	i586*
$\mathcal{H}^0_{\text{lin}}$ (var. 1)	5.4	2.7	1.2	1.2
$\mathcal{H}^0_{\text{lin}}$ (var. 2)	2.9	1.3	0.7	0.7
$\mathcal{H}_{\text{lin}}$ (var. 1)	5.4	2.7	1.2	1.2
$\mathcal{H}_{\text{lin}}$ (var. 2)	3.5	1.2	0.7	0.7
$\mathcal{H}_{\text{poly(3)}}$	6.2	2.4	1.6	1.6
$\mathcal{H}_{\text{mult}}$	1.0	0.3	0.2	0.2
$\mathcal{H}_{2\text{ tables}}$	1.0	0.6	0.8	1.6
$\mathcal{H}_{3\text{ tables}}$	0.5	0.1	0.3	0.4
$\mathcal{H}_{4\text{ tables}}$	0.3	0.1	0.3	0.3
$\mathcal{H}_{5\text{ tables}}$	0.4	0.1	0.3	0.3
$\mathcal{H}_{6\text{ tables}}$	0.7	0.2	0.3	0.3
$\mathcal{H}_{7\text{ tables}}$	0.8	0.2	0.4	0.3
$\mathcal{H}_{8\text{ tables}}$	0.9	0.3	0.4	0.4
$\mathcal{H}_{9\text{ tables}}$	1.0	0.3	0.4	0.4
$\mathcal{H}_{10\text{ tables}}$	1.0	0.3	0.5	0.5
$\mathcal{H}_{16\text{ tables}}$	1.6	0.5	0.8	0.8

[a]One hash function of each class was evaluated on 10^6 nonsequential keys. Listed is the average time per key in μs. For a description of the machines, see Section 3.1. "i586*" is an "i586" where the caches are disabled.

which differ in the way the mod p operation is implemented. Variant 1 uses the standard 64-bit mod operation. Variant 2 is based on the observation that one bit-shift and several 32-bit operations are sufficient if p is close to 2^{32} [Carter and Wegman 1979, p. 149]. This variant is faster than the first one. For the polynomial class $\mathcal{H}_{\text{poly(3)}}$, the mod p operation is implemented according to the second variant. Our implementation of the class $\mathcal{H}_{\text{mult}}$ exploits a property that all the processors used in our experiments share: in the case of an overflow during a 32-bit arithmetic operation, the processor computes the result modulo 2^{32}. Thus, 32-bit arithmetic suffices in $\mathcal{H}_{\text{mult}}$. The implementation of the table classes is described in Section 4.

To select the hash class for our implementation, we started with an *ad hoc* experiment where, for each class, one hash functions was fixed and the average time for the evaluation of this hash function was determined on a million keys. The keys are $ax \bmod 2^{32}$, $x = 1, \ldots, 10^6$, for some odd a (we used $a = 651{,}854{,}769$). Note that these keys are not random. However, they can be computed efficiently by the driver. Moreover, it seems reasonable to assume that none of the hash functions tested may take advantage of the pattern of the keys. This experiment was performed on machines with RISC processors (μSparcII, UltraSPARC-I) and on machines with CISC processors (Intel Pentium) processors (cf. Table III). The result shows that evaluating multiplicative, linear, or polynomial hash functions is expensive. The functions in the multiplicative class without primes and in the classes $\mathcal{H}_{t\text{ tables}}$ where t is small can be evaluated much faster. On RISC architectures (Sparc 5, Ultra 1), some of the classes $\mathcal{H}_{t\text{ tables}}$ are more efficient than $\mathcal{H}_{\text{mult}}$, while on the CISC architecture (i586), the class $\mathcal{H}_{\text{mult}}$ is more efficient. The high performance for $\mathcal{H}_{t\text{ tables}}$ on UltraSparc and on i586 is explained by the superscalar architecture of these platforms,

which makes it possible that more than one integer instruction per clock cycle is executed, and so they outperform the Sparc 5. Moreover, the higher number of registers provided by the RISC architecture makes it possible to store more intermediate results needed for the calculation of $\mathcal{H}_{t\ \text{tables}}$ in registers, so the UltraSparc beats the i586 for these classes by a wider margin than one would expect by comparing clock rates. One might expect a better performance for the class $\mathcal{H}_{\text{mult}}$ on the UltraSparc. However, the compiler was not targeted to use hardware multiplication instructions (which is available on the UltraSparc), so the compiled code uses software multiplication on both Sparc processors, while on the i586 hardware multiplication is used. This explains why the i586 outperforms the Sparc processors for the class $\mathcal{H}_{\text{mult}}$.

We decided to use the table class $\mathcal{H}_{4\ \text{tables}}$ for our implementation of the dictionary. To store a function from this class, four tables of size 256 are sufficient.

In the experiments described so far, the only task of the computer is to evaluate the hash function on one key after the other. This kind of experiment might give misleading results, because, for example, it is not clear whether pipelining influenced the performance of the arithmetic classes or how caching influenced the performance of the table classes.

Thus, a second experiment directed at the hash functions was performed after the implementation of the whole dictionary had been completed. The aim of this second experiment was to measure how fast the hash functions are evaluated when called in the context of the whole dictionary data structure. We describe this second experiment. We fix one level-1 and one level-2 hash function h_{test} and h'_{test} from the class to be tested. We use a modified implementation of the dictionary where the level-1 and level-2 hash functions are taken from the class $\mathcal{H}_{4\ \text{tables}}$ and where for each evaluation of one of these functions there is also an evaluation of h_{test} or h'_{test} on the same key. One million distinct keys are inserted into the dictionary. The keys are the same as in the *ad hoc* experiment. The size of the main table is 2^{22}.

Table IV contains the result of the second experiment. In each entry, the average total time needed for one successful lookup, including the additional evaluation of the level-1 hash function to be tested (as well as for the level-2 function, if required), is given. On the whole, the result of the second experiment confirms the result of the *ad hoc* experiment: It is most important to choose the hash class very carefully. If we implement dynamic perfect hashing with class $\mathcal{H}^0_{\text{lin}}$, $\mathcal{H}_{\text{lin}}$, or $\mathcal{H}_{\text{poly(3)}}$, the running time of the scheme may double. Hash functions from the classes $\mathcal{H}_{\text{mult}}$ and $\mathcal{H}_{t\ \text{tables}}$ are evaluated more efficiently. On the Ultra 1, functions from $\mathcal{H}_{4\ \text{tables}}$ are evaluated faster than functions from $\mathcal{H}_{\text{mult}}$, on the Sparc 5 the evaluation time is essentially the same, and on the i586 functions from $\mathcal{H}_{\text{mult}}$ are evaluated faster.

Looking only at the results for the Sparc 5, one sees how important it is to compare the evaluation time in a realistic environment, e. g., in the context of a dictionary implementation. While the *ad hoc* experiment (Table III, first column) suggests that the evaluation time of functions in $\mathcal{H}_{\text{mult}}$ and $\mathcal{H}_{2\ \text{tables}}$ is almost the same, the second experiment (Table IV, first column) reveals that our application functions from $\mathcal{H}_{\text{mult}}$ are, indeed, evaluated faster than those from $\mathcal{H}_{2\ \text{tables}}$. Also note that in contrast to the other

Table IV. Results of the Second Experiment[a]

	Sparc 5	Ultra 1	i586
$\mathcal{H}^0_{\text{lin}}$ (var. 1)	9.8 (9.74–9.81)	4.2 (4.16–4.35)	2.6 (2.57–2.61)
$\mathcal{H}^0_{\text{lin}}$ (var. 2)	8.0 (8.00–8.10)	2.9 (2.86–2.96)	2.1 (2.11–2.14)
$\mathcal{H}_{\text{lin}}$ (var. 1)	9.6 (9.48–9.80)	4.6 (4.38–5.45)	2.6 (2.60–2.63)
$\mathcal{H}_{\text{lin}}$ (var. 2)	7.8 (7.54–8.08)	2.9 (2.90–2.94)	2.2 (2.18–2.21)
$\mathcal{H}_{\text{poly(3)}}$	10.0 (9.96–10.02)	4.5 (4.44–4.64)	2.9 (2.89–2.93)
$\mathcal{H}_{\text{mult}}$	4.9 (4.88–5.10)	1.8 (1.75–1.80)	1.5 (1.47–1.50)
$\mathcal{H}_{2\text{ tables}}$	5.8 (5.67–5.79)	2.2 (2.19–2.33)	2.4 (2.37–2.40)
$\mathcal{H}_{3\text{ tables}}$	4.7 (4.69–4.81)	1.3 (1.34–1.38)	1.8 (1.82–1.88)
$\mathcal{H}_{4\text{ tables}}$	5.0 (4.89–5.01)	1.3 (1.32–1.39)	1.9 (1.81–1.89)
$\mathcal{H}_{5\text{ tables}}$	5.0 (4.86–5.00)	1.4 (1.36–1.76)	1.8 (1.80–1.91)
$\mathcal{H}_{6\text{ tables}}$	4.5 (4.44–4.55)	1.4 (1.39–1.57)	2.0 (1.93–2.03)
$\mathcal{H}_{7\text{ tables}}$	4.6 (4.59–4.69)	1.5 (1.43–1.88)	2.1 (2.09–2.11)
$\mathcal{H}_{8\text{ tables}}$	4.7 (4.68–4.81)	1.5 (1.47–1.56)	2.1 (2.09–2.20)
$\mathcal{H}_{9\text{ tables}}$	4.9 (4.86–4.96)	1.6 (1.50–1.96)	2.0 (2.01–2.11)
$\mathcal{H}_{10\text{ tables}}$	4.9 (4.89–4.92)	1.6 (1.53–1.87)	2.1 (2.03–2.09)
$\mathcal{H}_{16\text{ tables}}$	4.4 (4.40–4.43)	1.7 (1.73–1.77)	2.2 (2.19–2.22)

[a]Each entry is based on 100 trials each consisting of 10^6 successful lookups. The average operation time is listed in μs. The minimum and maximum values are given in parentheses. Note that on the Ultra 1 (because of swapping of other processes on the machine) the minimum and maximum values for $\mathcal{H}_{\text{lin}}$ (var. 1) and for some of the table classes differ significantly.

machines, on the Sparc 5 the most efficient one of the table classes uses $t = 16$ tables.

In both experiments, we observe a surprising behavior of the hash class $\mathcal{H}_{2\text{ tables}}$. Evaluation of functions from this class (two table lookups, one exclusive-or) takes more time than the evaluation of functions from other table classes, say from $\mathcal{H}_{4\text{ tables}}$ (four table lookups, three exclusive-or). Of course, the small tables used for the evaluation of functions in $\mathcal{H}_{4\text{ tables}}$ can be cached more efficiently than the larger ones used for $\mathcal{H}_{2\text{ tables}}$. Note that $\mathcal{H}_{4\text{ tables}}$ takes 4096 bytes (four tables of size 256, each entry is a 32-bit word), $\mathcal{H}_{2\text{ tables}}$ takes 524,288 bytes (two tables of size 65,536). Therefore, $\mathcal{H}_{4\text{ tables}}$ can be stored entirely in the L1-cache, while for an efficient calculation of $\mathcal{H}_{2\text{ tables}}$, the help of a L2-cache is needed. Thus, it seems reasonable to assume that the effect is because of caching.

If this hypothesis is true, then for those table classes where the table fits into the caches, the evaluation time should increase if the caches are disabled. (For $\mathcal{H}_{t\text{ tables}}$, $t = 7, 8, 9, 10, 16$, the total size of the tables is less than 1 KB.) For classes using large tables, the evaluation time without cache should not differ significantly. To test this hypothesis, we performed additional runs of the first experiment on machine i586, where the cache was disabled (see Table IV machine i586*). However, for the classes $\mathcal{H}_{t\text{ tables}}$, $t \geq 4$, the evaluation time does not depend on the presence of caches (i586 versus i586*). Moreover, the evaluation time for the classes $\mathcal{H}_{2\text{ tables}}$ and $\mathcal{H}_{3\text{ tables}}$ using large tables increases when the caches are disabled (0.80 versus 1.59 μs for $\mathcal{H}_{2\text{ tables}}$ and 0.28 versus 0.41 μs for $\mathcal{H}_{3\text{ tables}}$). We do not know how to explain this effect. The occurrence of this effect does not depend on the optimization option ($-$O0, ..., $-$O3) for the compiler.

```
extern unsigned int table[255*MAX_STR_LEN];

unsigned int string_hash(char *string){
    register const unsigned char *long_key = (const unsigned char *) string;
    register unsigned char last;
    register unsigned int hash_value=0, index=0;

    while((last=(unsigned char)*long_key++)!='\0'){
        hash_value^=table[index+=last];
    }
    return(hash_value);
}
```

Fig. 1. Implementation of Carter and Wegman's hash class for strings.

3.4 String Hashing

To obtain hash functions for strings (or, more general, keys of variable length), the following approach, because of Carter and Wegman, is used in our implementation.

Let the key be a sequence $c = c_1 c_2 \dots c_k$ of characters $c_i \in \{1, \dots, 255\}$. We assume that there is an upper bound MAX_STR_LEN on the length of the strings that may occur and that m is a power of 2. Each hash function h is described by a table of size $255 \cdot$ MAX_STR_LEN. The entries in this table are bit strings representing integers between 0 and $m-1$. To compute $h(c)$, the table entry at position $\sum_{j=1}^{i} c_j$ is read for each i, $1 \le i \le k$. The value $h(c)$ is the exclusive-or of all these entries (cf. Figure 1). Thus, $k-1$ additions, k table lookups, $k-1$ exclusive-or operations, and $k+1$ tests (versus '\0') suffice to compute $h(c)$. The class $\mathcal{H}_{\text{string}}$ consisting of all these functions is 1-universal.

4. IMPLEMENTATION

The dynamic perfect hashing scheme was implemented in C. In this section, we describe some details of our implementation. We also indicate where the implementation differs from the description in Section 2.

4.1 Efficient Representation of Small Buckets

To store n items in the dictionary, a main table of size at least $2n$ is used. Thus, most of the buckets in the table are small. (For $n = 2^{19}$ and main table size $m = 2^{20}$. In our implementation, more than 60% of the items are in buckets of size one, about 30% are in buckets of size two, and about 8% are in buckets of size three.)

We use a linked list rather than a subtable to store the items that are mapped to a small bucket. Since looking through a short list is more efficient than evaluating a level-2 hash function, the inserts, lookups, and deletes are performed faster. Another advantage is that lists use less space than subtables. Thus, the representation of the whole dictionary becomes more concise.

If a new item is hashed into a small bucket, where the items are stored in a list, it may be necessary to construct a subtable and to hash all items of the bucket into this subtable. If an item is deleted, the corresponding bucket may become small. Then, it is necessary to destroy the subtable and to put the

remaining items into a list. In our implementation, a subtable is constructed if the bucket stores at least three items. If, after a delete operation, only one item is left in the bucket, this item is put into a list (of length one). In this way, it will not happen that successive inserts and deletes of the same key cause the subtable to be repeatedly constructed and destroyed.

4.2 Lazy Resizing of Subtables

Subtables are used as long as possible, i.e., if an item is inserted into or deleted from a subtable, we try to use the same subtable and the same level-2 function to represent the new bucket. The subtable is resized only if a new item collides with an item already stored in this subtable.

4.3 Hash Functions

For each dictionary, one level-1 hash function and several level-2 hash functions are needed. As mentioned in Section 3, we use the 1-universal class $\mathcal{H}_{4\text{ tables}}$ for integer keys. For reasons of space, we can not store too many different such functions at the same time. Instead, we try to use the same level-2 function for different buckets. When the dictionary is constructed, a number of hash functions $h_1, h_2, \ldots, h_{hmax} \in \mathcal{H}_{4\text{ tables}}$ with $|U| = 2^{32}$ and range $\{0, \ldots, 2^{32} - 1\}$ is fixed. (In our implementation, we have $hmax = 64$.) The hash function for a subtable of size m_i or for the main table is represented by an index j, $1 \leq j \leq hmax$. To compute the function, h_j is evaluated modulo m_i. Thus, the hash functions can be stored in a concise way. When a new hash function is needed for some set of keys, the hash functions $h_{j+1}, h_{j+2}, h_{j+3}, \ldots$ are tested for being suitable one after the other; the first one that is suitable is selected. In each such situation, the probability that more than t functions are needed is bounded by 2^{-t}. In the unlikely event that the number $hmax$ is too small, new hash functions are selected and the whole dictionary is rehashed.

To hash string keys, the class $\mathcal{H}_{\text{string}}$ is used. We assume that the strings have, at most, 256 length. A table of $255 \cdot 256$ random numbers is needed to store one function from this class. In our implementation, one larger table of $255 \cdot 256 + hmax$ random numbers is used to represent $hmax = 4096$ hash functions for strings. The table, which defines hash function $h_j \in \mathcal{H}_{\text{string}}$, $1 \leq j \leq hmax$, is the subtable which starts at position j of this large table. Of course, these string hash functions h_j are not independent. Thus, we can not guarantee that the update operations in our implementation are executed in expected constant amortized time if the keys are strings.

To initialize the tables for the hash functions, a large table of random numbers fixed at compile time is used.

4.4 Rehashing

There are several reasons that may cause a rehashing of the whole dictionary. The most frequent reason is that the size of the main table is changed. In this case, we can use the same function h_j as for the old main table. To improve the performance of rehashing, at the time of insertion (as well as during each rehash where a new function $h_{j'}$ is chosen), for each item with key component

x, we compute and store the value $h_j(x)$. The value of the new level-1 hash function is just the stored value $h_j(x)$ modulo the new table size.

For the level-1 hash function, we require that the sum of the squares of the bucket sizes is always bounded by $2n$.

4.5 Size of the Tables

The size of all tables is a power of 2. Initially, the main table has size $m = 2^{16}$. If the number of elements in the dictionary becomes larger than $m/2$, then m is quadrupled. We do not change the size of the main table if items are deleted. Thus, the size of the main table depends on the maximum number of items that have ever been in the dictionary at the same time.

If a new subtable for a bucket of size i is to be constructed, the size of the subtable is the smallest power of 2 that is at least $2\,i^2$. The size of the subtable is changed when it is rehashed.

The tables contain pointers to the items. Thus, it is not necessary to copy the items during a rehashing. This also makes the implementation persistent in the sense of McGeoch [1996b]. We remark that to represent an empty dictionary, approximately 680 KB of memory are used. (Each of the 2^{16} entries of the main table uses 6 bytes; 280 KB are used to represent the hash functions.)

4.6 Memory Allocation

We use the standard memory allocation routines of the malloc library. To improve the performance of these routines, two well-known strategies are used. (1) Memory for items is allocated in blocks. For each 10,000 items, memory is allocated at once. (2) If an item is deleted, its memory is not deallocated. Instead, a pointer to this memory location is entered to a separate free list. This memory is used when a new item is inserted. Free lists are also used to avoid deallocation of the memory used for subtables.

5. EXPERIMENTS

5.1 Test Driver

Our test driver complies with the input file specifications described in McGeoch [1996b]. In the input files, the data is in ASCII. The driver reads the input file, converts the ASCII data into C data, and calls the appropriate dictionary operations.

However, the `scanf` command used to convert the data is a very expensive C operation. Most of the CPU time is spent for conversion. In order to compare the implementations of the dictionary (rather than to compare implementations of `scanf` on different machines) and to get more realistic measurements, our test driver consists of two parts. In the first part, the whole input file is read into a large array. Each entry of the array stores one command, i.e., the type of the operation and the key and information component of the item. In the second part of the driver, this array is used to call the dictionary operations. Some of the input files are too large to fit into the main memory of our machines. For these files, a modified driver is used, which consists of two programs. The first

program converts the data and pipes it to the second program, which calls the dictionary operations. Our test drivers do not output the result of dictionary operations.

The drivers and the implementations were compiled with the GNU gcc compiler v2.7.2 with optimization option -O2. (For the experiments in Section 3, gcc v2.7.0 with optimization option -O3 was used.)

5.2 Results

In the following, we describe the results of tests with our implementation (`dyn_perf`) and with the demo dictionary implementation provided by DIMACS (`dc_demo`).

We use the UNIX `getrusage` routine to measure the CPU time used by the process. This routine measures the user and the system time of the process. The clock tick interval is $\leq$1000 μs, on the Ultra, and 10,000 μs, on the other machines. All measurements refer to the sum of the user and the system time *spent in the second part of the driver*, i.e., only the time needed to perform the dictionary operations, as well as the time needed to construct and destroy the dictionary is measured. In the tables, for each input file and each machine, we state the average time for one dictionary operation (user plus system time, divided by the number of commands (in μs)). Because of the large clock tick interval, at most, the first two digits of the entries are significant.

5.2.1 *Performance of Basic Dictionary Operations.* First, we measure the time needed for the basic dictionary operations (construct, insert, lookup, delete, and destroy) and for the rehashing of the dictionary. The keys are random integers. The information component is omitted. We use special test files and measure only the time needed for lookups (or deletes, etc.).

There is a large overhead for the construction of a new dictionary. On the Sparc 10, Sparc 5, and the Ultra 1, the construction of an empty dictionary and the insertion of the first item together take approximately 180,000, 150,000, or 29,000 μs, respectively. This may be a problem for very small dictionaries with only a few thousand operations. Our implementation should be used only if the number of operations is not too small.

Table V contains the results for the other dictionary operations. In dynamic perfect hashing, the performance of the insert operation depends on whether the main table has to be resized or not. Column "insert (1)" in Table V covers the case where the items are inserted into a new dictionary, which was just created. In this case, the main table has to be resized and the items have to be rehashed upon the insert of item number 32,769 and 131,073. Column "insert (2)" covers the case that the dictionary is empty, but has been used before. Here, the main table is already large enough such that the items are inserted without rehashing and without allocating additional memory. Note that in case (1), the average time for the insertion of 131,072 items is smaller than the average time for the insertion of 65,536 items. The reason is that, in both instances, the main table size is quadrupled exactly once. This causes a rehashing of 32,768 items. (The same holds for 524,288 versus 262,144 insertions. In these instances, the main table size is quadrupled twice.) In case (2), there are no rehashes. Thus,

Table V. Performance of Basic Dictionary Operations for Integer Keys on the Ultra 1[a]

No. of Items	Insert (1)	Insert (2)	Rehash
65,536	3.5 (3.45–3.57)	1.5 (1.46–1.50)	2.3 (2.29–2.35)
131,072	3.0 (2.98–3.00)	1.7 (1.63–1.85)	2.2 (2.22–2.29)
262,144	4.2 (4.20–4.24)	1.5 (1.53–1.59)	2.6 (2.53–2.62)
524,288	3.5 (3.39–4.45)	1.9 (1.87–2.06)	2.4 (2.39–2.43)

No. of Items	Lookup (successful)	Lookup (unsuccessful)
65,536	1.2 (1.19–1.19)	1.1 (1.11–1.13)
131,072	1.3 (1.31–1.31)	1.3 (1.29–1.30)
262,144	1.3 (1.27–1.28)	1.2 (1.19–1.19)
524,288	1.4 (1.37–1.46)	1.5 (1.41–1.96)

No. of Items	Delete	Destroy
65,536	1.3 (1.30–1.31)	1.0 (0.96–0.98)
131,072	1.5 (1.46–1.50)	1.0 (1.04–1.05)
262,144	1.4 (1.38–1.39)	1.1 (1.10–1.11)
524,288	1.5 (1.53–1.69)	1.2 (1.16–1.19)

[a]Each entry is based on 40 trials. The average operation time is listed in μs. The minimum and maximum values are given in parentheses.

the average time of the inserts is much smaller. Observe that in case (2) the average time for 131,072 inserts (into a main table of size 262,144) is a bit larger than the corresponding time for 65,536 inserts (main table of size 262,144) or 262,144 inserts (main table size 1,048,576). The reason is that the insertion of a new item is performed faster if the load factor of the main table is small.

The column "rehash" in the table refers to a subroutine with the same name. When this routine is called, a new level-1 hash function is chosen. The size of the main table is not changed. The entries in this column show that rehashing the whole table is expensive. This explains why we quadruple (rather than double) the main table if it becomes too small.

A comparison of the time needed for the insert (2) operation and of the time needed for the lookup operation shows that there is only a small overhead for building and rehashing the subtables. This is in accordance with our expectation, since in our implementation only few buckets are represented by subtables.

5.2.2 *DIMACS Test Inputs.* The Tables VI–VIII contain the results with test inputs provided by DIMACS. Tables IX and X contain tests with inputs provided by Craig Silverstein and Darko Stefanovic, two participants of the DIMACS Challenge. In all tests, the information component is omitted.

5.2.2.1 *Random Numeric Key Tests (Table VI).* The keys in these inputs are integers. The experiments illustrate the influence of the construction time on the average performance of the operations. On test files of size "b" (20,000–40,000 lines), a substantial part of the computation time of `dyn_perf` is used to construct the dictionary. The test files of size "c" show how this influence declines when the number of operations increases.

5.2.2.2 *Generic English Key Tests (Table VII).* Here, the keys are strings of length at most 20. The performance of `dyn_perf` on the files `ed.*` is impaired

Table VI. Comparison of the Performance of Dynamic Perfect Hashing (dyn_perf) and of the Demo Dictionary Implementation Provided by DIMACS (dc_demo)[a]

Input File	No. of Lines	Sparc 10 dyn_perf	Sparc 5 dyn_perf	Ultra 1 dyn_perf
id.b	20,009	15.0 (14.5–17.5)	11.1 (10.5–13.0)	2.9 (2.9–3.0)
iid.b	30,009	12.3 (11.7–14.0)	9.0 (8.3–9.3)	2.5 (2.5–2.6)
iisd.b	40,009	11.0 (10.2–12.0)	7.6 (7.5–8.7)	2.3 (2.2–2.3)
iiud.b	40,009	10.8 (10.5–12.0)	7.6 (7.5–8.2)	2.3 (2.2–2.3)
is.b	20,009	15.0 (14.5–17.0)	11.1 (11.0–12.0)	3.0 (2.9–3.0)
iu.b	20,009	15.1 (15.0–16.5)	11.0 (9.5–11.5)	2.9 (2.9–3.0)
id.c	200,009	12.3 (12.2–12.7)	8.1 (7.9–8.4)	2.9 (2.8–2.9)
iid.c	300,009	11.9 (11.7–13.1)	7.6 (7.6–7.8)	2.9 (2.8–3.0)
iisd.c	400,009	10.8 (10.5–11.5)	6.6 (6.6–6.7)	2.7 (2.6–2.7)
iiud.c	400,009	10.6 (10.4–11.2)	6.6 (6.5–6.7)	2.7 (2.7–2.7)
is.c	200,009	13.0 (12.9–13.2)	8.3 (8.1–8.6)	3.2 (3.1–3.2)
iu.c	200,009	13.0 (12.9–13.2)	8.3 (8.1–8.5)	3.2 (3.1–3.2)
Input File	No. of Lines	Sparc 10 dc_demo	Sparc 5 dc_demo	Ultra 1 dc_demo
id.b	20,009	29.5 (29.0–33.5)	18.7 (17.0–20.0)	4.6 (4.6–4.8)
iid.b	30,009	29.0 (27.3–30.7)	18.7 (18.7–19.0)	4.7 (4.6–4.8)
iisd.b	40,009	26.6 (25.2–28.2)	17.3 (16.5–18.5)	4.3 (4.2–4.4)
iiud.b	40,009	26.1 (24.7–30.0)	17.5 (16.5–18.5)	4.4 (4.3–4.4)
is.b	20,009	29.1 (28.5–29.5)	18.8 (18.0–20.0)	4.8 (4.6–4.9)
iu.b	20,009	29.3 (29.0–30.0)	19.2 (18.5–20.0)	4.8 (4.7–5.0)
id.c	200,009	47.6 (46.6–48.2)	26.9 (25.7–29.4)	8.8 (8.3–9.2)
iid.c	300,009	48.6 (47.1–51.4)	27.9 (26.5–29.4)	9.1 (8.9–9.4)
iisd.c	400,009	45.2 (43.5–47.2)	26.2 (24.1–27.6)	9.1 (9.0–9.6)
iiud.c	400,009	43.7 (43.1–44.4)	25.7 (24.1–27.1)	8.9 (8.9–9.0)
is.c	200,009	43.8 (43.6–44.1)	25.7 (25.5–25.8)	8.0 (7.8–8.1)
iu.c	200,009	43.9 (43.5–44.3)	26.5 (25.5–27.7)	8.0 (7.8–8.3)

[a]The test inputs are traces from dc_random (Random numeric key tests). Each entry is based on 30 trials. The average operation time is listed in μs.

because the number of commands in these files is small. We built larger test files word.* using the Unix dictionary (/usr/dict/words). These files have the same structure as the corresponding files in Table VI. Thus, the Tables VI and VII display a slight difference in the performance of integer hashing and of hashing short strings. Of course, dc_demo performs badly on sorted keys (>1 ms). Thus, results of measurements of dc_demo on word.* files are not listed in Table VII.

5.2.2.3 *Wordcount Problems (Table VIII).* The keys are strings of length, at most, 20. Because of the size, only the files eddington.dat and joyce.dat yield meaningful results. Most of the commands in these files are lookups; only few items are inserted. The experiments show that the hash functions in the class $\mathcal{H}_{\text{string}}$ can be used to build efficient dictionaries.

5.2.2.4 *Traces from Library Call Numbers (Table IX).* These files contain larger keys (up to 36 characters) and much more insert operations than the files from dc_wordcount. The experiments show that the average execution time of

Table VII. Traces from dc_generic (Generic English Key Tests)[a]

Input File	No. of Lines	Sparc 10 dyn_perf	Sparc 5 dyn_perf	Ultra 1 dyn_perf
ed.id.tr	2007	105.6 (99.7–120)	83.2 (79.7–84.7)	18.3 (18.2–18.7)
ed.iid.tr	3007	72.5 (69.8–73.2)	57.2 (56.5–59.9)	13.3 (13.3–13.4)
ed.iisd.tr	4007	57.4 (57.4–57.4)	44.4 (42.4–44.9)	10.2 (10.1–10.4)
ed.iiud.tr	4007	57.4 (54.9–67.4)	45.7 (42.4–54.9)	10.1 (10.1–10.2)
ed.is.tr	2007	103.6 (99.7–105)	81.7 (79.7–84.7)	18.1 (18.0–18.2)
ed.iu.tr	2007	104.1 (99.7–105)	83.2 (79.7–84.7)	18.3 (18.1–18.5)
word.id.tr	20,007	19.5 (19.0–21.0)	13.8 (13.0–14.5)	4.0 (3.9–4.0)
word.iid.tr	30,007	18.8 (17.7–20.0)	12.8 (12.0–13.3)	4.0 (3.9–4.1)
word.iisd.tr	40,007	16.1 (15.2–16.7)	10.8 (10.5–12.0)	3.4 (3.4–3.4)
word.iiud.tr	40,007	15.7 (15.5–16.0)	10.5 (9.7–11.5)	3.3 (3.3–3.4)
word.is.tr	20,007	19.3 (18.0–20.5)	13.5 (13.0–15.0)	3.8 (3.8–3.8)
word.iu.tr	20,007	18.8 (17.5–19.5)	13.3 (13.0–15.0)	3.7 (3.7–3.7)
Input File	No. of Lines	Sparc 10 dc_demo	Sparc 5 dc_demo	Ultra 1 dc_demo
ed.id.tr	2007	44.3 (39.9–44.8)	27.4 (24.9–29.9)	11.5 (11.3–11.6)
ed.iid.tr	3007	41.6 (39.9–43.2)	26.6 (26.6–26.6)	11.2 (10.9–11.4)
ed.iisd.tr	4007	37.4 (37.4–37.4)	23.5 (22.5–25.0)	11.2 (11.1–11.4)
ed.iiud.tr	4007	40.7 (39.9–42.4)	25.0 (25.0–25.0)	11.6 (11.6–11.7)
ed.is.tr	2007	45.8 (44.8–49.8)	27.4 (24.9–29.9)	12.3 (12.1–12.5)
ed.iu.tr	2007	40.9 (34.9–44.8)	27.9 (24.9–29.9)	12.3 (12.1–12.5)

[a]Ten trials for each entry. The average operation time is listed in μs.

Table VIII. Traces from dc_wordcount (Word Count Problems)[a]

Input File	No. of Lines	Sparc 10 dyn_perf	Sparc 5 dyn_perf	Ultra 1 dyn_perf
alice.dat	736	269.0 (258.2–312.5)	210.6 (203.8–217.4)	45.5 (45.4–45.7)
boorstin.dat	571	336.3 (332.7–350.3)	280.2 (262.7–297.7)	58.8 (58.5–59.1)
declaration.dat	1862	109.0 (107.4–112.8)	87.5 (85.9–91.3)	18.7 (18.6–18.9)
eddington.dat	75,027	7.7 (7.6–8.1)	5.2 (4.9–5.3)	1.4 (1.4–1.4)
joyce.dat	93,220	7.4 (7.4–7.5)	4.9 (4.7–4.9)	1.5 (1.5–1.5)
twain.dat	617	319.3 (307.9–372.8)	252.8 (243.1–259.3)	54.2 (54.0–54.5)
Input File	No. of Lines	Sparc 10 dc_demo	Sparc 5 dc_demo	Ultra 1 dc_demo
alice.dat	736	24.5 (13.6–27.2)	14.9 (13.6–27.2)	7.7 (7.6–8.0)
boorstin.dat	571	29.8 (17.5–35.0)	17.5 (17.5–17.5)	7.6 (7.5–7.7)
declaration.dat	1862	24.2 (21.5–26.9)	16.6 (16.1–21.5)	8.4 (8.2–8.5)
eddington.dat	75,027	22.3 (22.1–22.7)	14.2 (14.1–14.4)	10.2 (10.1–10.3)
joyce.dat	93,220	25.7 (25.6–25.9)	16.1 (16.0–16.2)	10.1 (10.1–10.2)
twain.dat	617	25.9 (16.2–32.4)	16.2 (16.2–16.2)	7.8 (7.6–7.9)

[a]Ten trials for each entry. The average operation time is listed in μs.

the commands in these files is about three times larger than for the large files from dc_wordcount.

5.2.2.5 *Traces from Object References (Table X).* The keys are integers. Only few insert operations are contained in these files. For the "smalltalk" files, the maximum size of the dictionary is rather small (about 2000–60,000 items). With the exception of file trace-3.11-Q-1, for the other files the maximum size of the dictionary is less than 200,000.

Table IX. Traces from Library Call Numbers[a]

Input File	No. of Lines	Sparc 10 dyn_perf	Sparc 5 dyn_perf	Ultra 1 dyn_perf
circ1.10	36	5777.8 (5556–6667)	4638.9 (4167–5556)	974.5 (967–1005)
circ1.100	226	898.2 (885–929)	805.3 (752–1062)	158.1 (158–159)
circ1.1000	1936	115.7 (114–119)	91.9 (87.8–93.0)	20.8 (20.7–20.9)
circ1.10000	20,106	22.2 (21.4–24.4)	15.8 (15.4–16.4)	4.6 (4.6–4.6)
circ2.10000	19,974	21.7 (21.5–23.0)	15.6 (15.5–16.0)	4.7 (4.6–4.7)
circ3.10000	20,298	21.6 (21.2–22.2)	15.6 (15.3–15.8)	4.7 (4.7–4.7)
circ4.10000	20,394	21.5 (21.1–22.6)	15.4 (15.2–15.7)	4.6 (4.6–4.6)
circ5.10000	19,964	21.8 (21.5–22.0)	15.7 (15.5–16.0)	4.7 (4.7–4.7)
circ1.100000	198,786	19.2 (19.1–19.3)	12.6 (12.6–12.7)	4.8 (4.6–5.0)
circ2.100000	200,014	19.0 (18.9–19.1)	12.5 (12.4–12.5)	4.6 (4.6–4.6)
circ3.100000	200,784	19.2 (19.1–19.3)	12.6 (12.6–12.8)	4.6 (4.5–4.6)
circ4.100000	199,954	19.1 (19.0–19.3)	12.5 (12.5–12.6)	4.6 (4.6–4.6)
circ5.100000	200,256	19.2 (19.1–19.3)	12.6 (12.5–12.6)	4.6 (4.6–4.7)
Input File	No. of Lines	Sparc 10 dc_demo	Sparc 5 dc_demo	Ultra 1 dc_demo
circ1.10	36	0.0 (0.0–0.0)	0.0 (0.0–0.0)	7.9 (7.7–8.1)
circ1.100	226	35.4 (0.0–44.2)	4.4 (0.0–44.2)	7.5 (7.3–7.7)
circ1.1000	1936	49.6 (46.5–51.7)	30.5 (25.8–31.0)	9.6 (9.4–9.8)
circ1.10000	20,106	65.1 (64.7–65.7)	39.9 (39.8–40.3)	13.2 (13.2–13.6)
circ2.10000	19,974	67.4 (67.1–68.1)	41.4 (41.1–42.1)	13.6 (13.5–13.7)
circ3.10000	20,298	63.3 (63.1–63.6)	39.1 (38.9–39.4)	12.8 (12.7–12.8)
circ4.10000	20,394	63.8 (63.7–64.2)	39.3 (38.7–39.7)	13.0 (12.9–13.0)
circ5.10000	19,964	66.2 (66.1–66.6)	41.3 (40.6–44.1)	13.5 (13.4–13.9)
circ1.100000	198,786	89.7 (89.6–89.8)	55.6 (55.1–55.8)	20.2 (20.1–20.4)
circ2.100000	200,014	91.7 (91.6–92.0)	56.1 (55.8–56.3)	20.3 (20.2–20.5)
circ3.100000	200,784	88.0 (87.9–88.2)	53.2 (52.9–53.4)	19.1 (19.0–19.3)
circ4.100000	199,954	86.9 (86.8–87.3)	53.0 (52.6–53.3)	19.0 (19.0–19.1)
circ5.100000	200,256	93.3 (93.2–93.5)	56.5 (56.1–56.7)	20.5 (20.5–20.7)

[a]Ten trials for each entry. The average operation time is listed in μs. Data provided by Craig Silverstein.

6. VARIATION OF PARAMETERS

In this section we analyze how the choice of several parameters of the dictionary influences the runtime behavior.

First, we want to find a good choice for switching between subtables and lists as representation of the buckets. Second, we vary the *load factor* α, which is defined as

$$\alpha := \frac{n}{m}.$$

Finally, we compare how much the resizing factor of the main table changes the runtime.

6.1 Choosing the Minimal Size of a Subtable

In our implementation, we use lazy subtable creation. Let k be the largest number of items in a bucket represented as a list. In a series of experiments, we tried to find out how the choice of k influences the runtime behavior of the dictionary.

For the first experiment, the input file consisted only of inserts of 57,959 different strings (file `ins.dat`). The strings were taken from the *Revidierte*

Table X. Traces from Object References[a]

Input File	No. of Lines	Sparc 10 dyn_perf	Sparc 5 dyn_perf	Ultra 1 dyn_perf
trace-3.11-Q-1	20,169,760	21.1(20.8–21.5)	—	—
trace-3.16-P	49,174,317	7.2 (7.2–7.3)	4.4 (4.4–4.5)	1.7 (1.7–1.7)
trace-3.16-Q	34,383,361	6.8 (6.7–7.0)	4.4 (4.4–4.4)	1.7 (1.7–1.8)
trace-3.2-Y-1	1,326,857	7.5 (7.4–7.7)	4.9 (4.7–5.1)	1.8 (1.8–1.8)
trace-4.16-P	49,211,767	7.2 (7.0–7.3)	4.5 (4.4–4.5)	1.9 (1.7–2.5)
trace-4.16-Q	34,422,746	7.0 (6.9–7.2)	4.5 (4.4–4.5)	1.7 (1.7–1.7)
trace-Smalltalk-19R	21,809,927	6.5 (6.4–6.6)	4.1 (4.0–4.1)	1.5 (1.5–1.5)
trace-Smalltalk-2R	14,585,336	6.7 (6.6–6.8)	4.2 (4.1–4.2)	1.5 (1.5–1.5)
trace-Smalltalk-2	2,134,345	6.9 (6.9–7.0)	4.7 (4.6–4.7)	1.7 (1.7–1.7)
trace-Smalltalk-3R	5,435,940	6.6 (6.4–6.7)	4.0 (4.0–4.1)	1.5 (1.5–1.5)
trace-Smalltalk-3	757,454	7.0 (6.9–7.2)	4.8 (4.8–4.9)	1.6 (1.6–1.6)
trace-Smalltalk-4R	6,314,566	6.4 (6.2–6.6)	4.2 (4.1–4.2)	1.5 (1.5–1.6)
trace-Smalltalk-4	1,158,384	6.8 (6.6–7.1)	4.6 (4.4–4.7)	1.8 (1.6–2.4)
trace-Smalltalk-5R	5,949,391	6.3 (6.2–6.3)	4.0 (4.0–4.0)	1.4 (1.4–1.4)
trace-Smalltalk-5	889,951	6.5 (6.1–6.7)	4.2 (4.1–4.4)	1.5 (1.5–1.5)
trace-Smalltalk-Richards	3,314,690	6.1 (6.0–6.3)	4.2 (4.2–4.3)	1.4 (1.4–1.4)
trace-Smalltalk-Session1	95,416,787	6.8 (6.8–6.8)	4.2 (4.1–4.2)	1.8 (1.5–2.3)

[a]Five trials for each entry. The average operation time is listed in μs. For the file trace-3.11-Q-1, the program dyn_perf required about 170 MB of space and caused heavy swapping on the Sparc 10. The tests on the Sparc 5 and the Ultra 1 were cancelled, because they did not complete in reasonable time. Data provided by Darko Stefanovic.

Elberfelder Bibel (http://www.joyma.com/elberfe.htm), including all chapter numbers, references, etc. In the second experiment, we appended one lookup for each key to the input file (file inslkp.dat). In both experiments, we varied k and measured the average CPU time for execution of an operation. If a_k is the average cost of an (insert) operation in the first and b_k the average cost of an operation in the second experiment, we can find an approximate value of the average cost of a lookup by calculating $2b_k - a_k$. In this way, we calculated the average time for a lookup. The results can be seen in Figure 2. We used several machines, which differ in architecture and processor type. As we can see, the curves look similar on different machines.

We see that the cost of an insert decreases for $k = 0, 1, 2, 3$ and stays constant for larger k. Because there are only few buckets of size 3 and larger this result is no surprise: If $k \geq 4$ there are extremely few buckets left, which are rebuilt to a subtable. For $k \geq 1$ the cost of a lookup is constant; it is larger for $k = 0$ when for a singleton buckets a level-2 hash function is evaluated. Although there are numerous buckets of size 2 and 3, we can not see any difference in the amortized cost of a lookup for $k = 1, 2, 3$. It seems that the operations that are needed to identify an item take similar time in lists ($\frac{k}{2}$ comparisons + $\frac{k}{2} - 1$ jumps to the next item) and in subtables (one calculation of the hash function + one comparison).

In conclusion, we see that in our implementation of the dictionary as a 2-level hash table from the amortized point of view, the use of a hash function for the second step is more expensive than using simple lists. This holds as long as α is small. Note that in our implementation the main table is resized when α reaches 1/2.

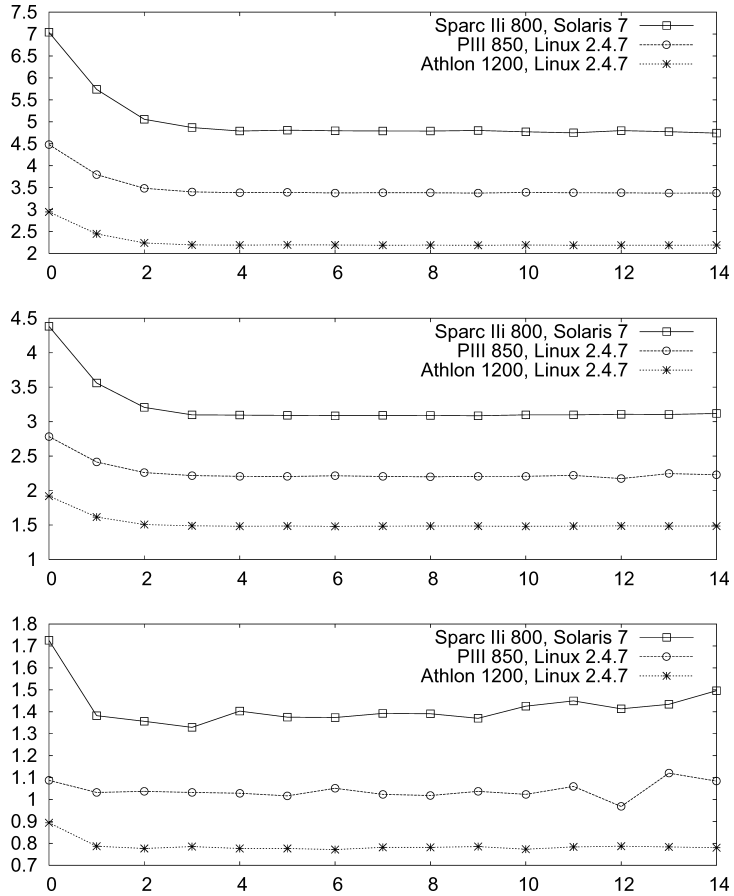

Fig. 2. CPU times for inserts, inserts + lookups, and lookups for varying maximal chain length k. The x axis represents k and the y axis represents the average CPU time (in μs) needed for an insert (top), an insert and lookup (middle), and a lookup (bottom) operation.

6.2 The Load Factor

In this experiment, we wanted to find out what happens when α gets larger. Thus, we changed the dictionary implementation at this point: Instead of quadrupling the size m of the main table when $m/2$ items are in the dictionary, now m is quadrupled when an item is inserted and α becomes larger than a given value $\alpha_{\max}$.

The expected sum of squares of bucketsizes is now (see Equation (1))

$$E_{h\in\mathcal{H}_4 \text{ tables}}\left(\sum_{i=0}^{m-1} b_{h,i}^2\right) \le n + \frac{n(n-1)}{n/\alpha} \le n(1+\alpha) \tag{2}$$

Therefore we can bound the sum of the squares of the bucket sizes by $(1+\alpha_{\max})n$.

We repeated the last experiments with several values for $\alpha_{\max}$. The average CPU time for an lookup is given in Figure 3 and for an insert in Figure 4, each

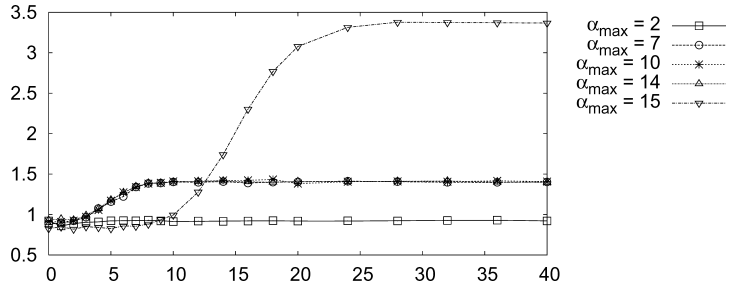

Fig. 3. The average CPU time for lookups with different load factors. The x axis represents k and the y axis represents the average CPU time (in μs) needed for a lookup operation.

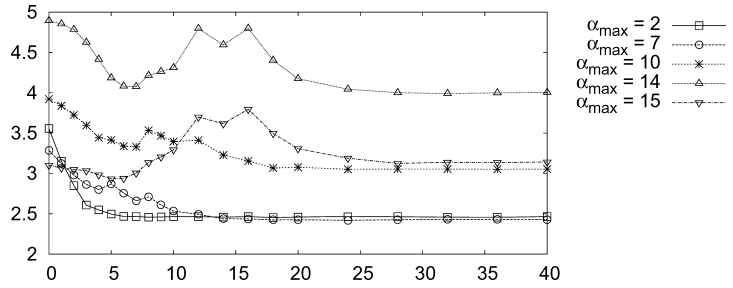

Fig. 4. The average CPU time for inserts with different load factors. The x axis represents k and the y axis represents the average CPU time (in μs) needed for an insert operation.

for $\alpha_{\max} = 2, 7, 14, 15$. For this and the next experiments, we just used the fastest machine from the last experiment.

One advantage of a higher load factor is the reduction of the number of complete rehashes. If we expect r rehashes of the main table for a given input file if $\alpha_{\max} = \alpha_0$, then we expect $r - 1$ rehashes, if $\alpha_{\max} = 4\alpha_0$. We can observe this effect, if we compare the average runtimes in Figure 4 for $\alpha_{\max} = 14$ and $\alpha_{\max} = 15$, where the extra rehash needed for $\alpha_{\max} = 14$ increases the average runtime significantly.

Another possible effect (especially for a consecutive series of lookups) is a better cache hit rate for accesses to the (smaller) main table. For this improvement, we pay with a large amount of memory needed by the the level-2 subtable, because the expected amount of memory for it grows linearly in α.

We observe a large difference in the the lookup time for large chains for $\alpha_{\max} = 14$ and $\alpha_{\max} = 15$, which is obviously the result of larger buckets α for $\alpha_{\max} = 15$. In the following we explain this dramatic change.

If k is arbitrarily large, then, for the average CPU time needed for a lookup operation, the average number of items in lists is the most important factor. This depends on the number of lists and on the final load factor α, because the expected bucket size is α. The final load factor α depends on $\alpha_{\max}$ and on the the number of items in the dictionary.

After all items have been inserted into the dictionary, the value of α always ranges between $\alpha_{\max}/4$ and $\alpha_{\max}$. To get the exact value of α we must find the

final size of the main table. In this series of experiments, we start with a main table size of $m = 1024$. If we take $\alpha_{\max} = 14$, the main table is quadrupled twice to ensure $\alpha < \alpha_{\max}$. Thus, we finally get $\alpha = 57,959/16{,}384 \approx 3.54$. If $\alpha_{\max} = 15$, the main table is quadrupled only once and we get $\alpha \approx 14.15$.

We can also see that we get the same curves for $\alpha_{\max} = 7, 10, 14$. The explanation is simple: We get the same α for each $\alpha_{\max}$.

Another effect we observe: Let $\alpha_{\max}$ be fixed and let $k > 3$ grow. The average CPU time for a lookup operation then also grows. A larger k causes more comparisons with items stored in lists to find a given key. In the diagram for $\alpha_{\max} = 15$, we have four times faster lookup operations if we use $k \leq 8$ compared with $k \geq 24$.

If the input were random or the hash function were fully random, we would expect that most of the buckets have a size close to α. This is why in Figure 3 we see the largest slope in the curve for $\alpha_{\max} = 14$ between $k = 3$ and $k = 4$ and for $\alpha_{\max} = 15$ between $k = 14$ and $k = 15$.

Finally, note that if $k \leq 12$ the lookup time is smaller for $\alpha_{\max} = 14$ than for $\alpha_{\max} = 15$ and, vice versa, if $k > 12$. The reason is that for $\alpha_{\max} = 14$, nearly all buckets are organized as lists, but if $\alpha_{\max} = 15$ only a few are. If k grows and $k > 12$, for $\alpha_{\max} = 15$ the number of lists grows as well, but these lists are about four times longer than if $\alpha_{\max} = 14$ and so the runtime behavior changes.

We get two results of this experiment concerning lookup operations: First, the representation of large buckets as lists is not a good choice and, second, the average lookup time is not a monotonic function of $\alpha_{\max}$.

If we look at the time for insert operations, we observe, in general, that for growing k the average CPU time for an operations decreases. The reason is the large cost of creating a subtable. The larger k is chosen as the fewer lists are transformed into a subtable. The only exception of this rule are some strange peaks that occur for large $\alpha_{\max}$ as a result of two things that happen if an item is inserted into the dictionary: First the item is searched in the dictionary. If not found, the item is inserted into a bucket, which may result in a birth of a new subtable. While the search is faster for smaller k, the creation of the subtables is more expensive for smaller k because more buckets have to be reorganized as subtables. The resulting plots essentially show the superposition of the first increasing and the second decreasing function.

6.3 The Resizing Factor

If we change the resizing factor from 4 to 2, we expect larger buckets and larger load factors, on average, and about twice as many rehash operations. This may result in higher cost for the operations. To see how a change of the resizing factor affects the running time, we run a series of experiments for which input files from the DIMACS challenge website were used. We experimented with different instance types (`eddington.dat` and `joyce.dat` for strings as instance types and `matchexact.trace-3.11-Q-1` and `matchexact.trace-Smalltalk-Session1` for integers) and different load factors ($\alpha_{\max} = 1, 10, 20$). We varied k and measured the average CPU time for an operation (Figures 5–8).

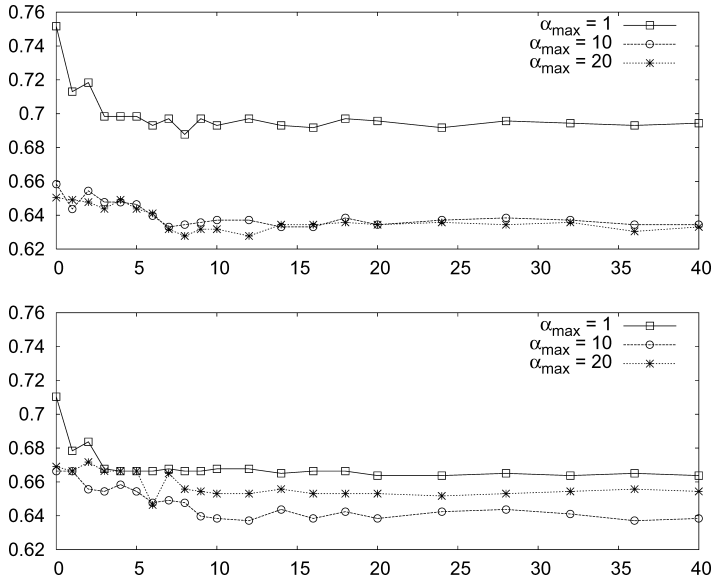

Fig. 5. The average CPU time for one operation for `eddington.dat`. (Top) Plots for resizing factor 2; (bottom) Plots for resizing factor 4. The x axis represents k and the y axis represents the average CPU time (in μs) needed for an operation.

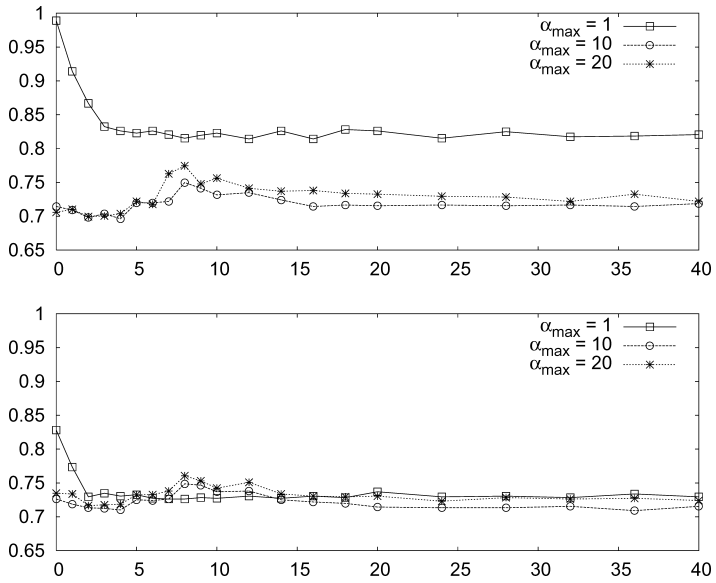

Fig. 6. The average CPU time for one operation for `joyce.dat`. (Top) Plots for the resizing factor 2; (bottom) Plots for resizing factor 4. The x axis represents k and the y axis represents the average CPU time (in μs) needed for an operation.

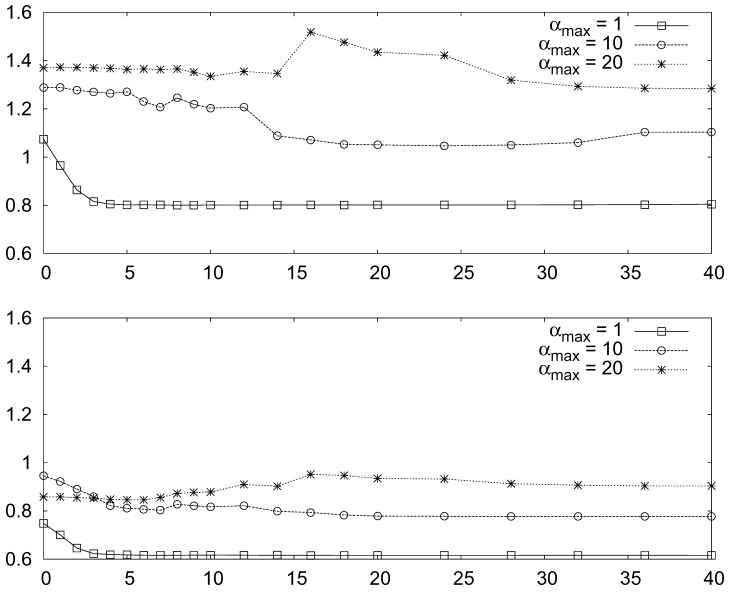

Fig. 7. The average CPU time for one operation for `matchexact.trace-3.11-Q-1`. (Top) Plots for the resizing factor 2; (bottom) Plots for the resizing factor 4. The x axis represents k and the y axis represents the average CPU time (in μs) needed for an operation.

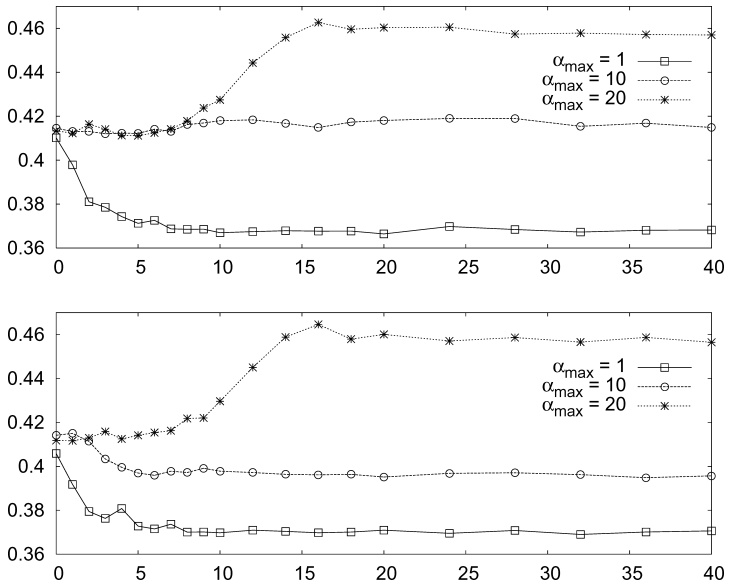

Fig. 8. The average CPU time for one operation for `matchexact.trace-Smalltalk-Session1`. (Top) Plots for resizing factor 2; (bottom) Plots for resizing factor 4. The x axis represents k and the y axis represents the average CPU time (in μs) needed for an operation.

Obviously, the runtime performance is only slightly better if we use a resizing factor 4. This holds for both integer and string instances and for different load factors. The reason are fewer rehashes and for larger k shorter lists. The largest difference we observe for the file `matchexact.trace-3.11-Q-1`, which creates a very large dictionary and has the largest ratio of number of inserts to number of lookups; this suggests that the resizing factor has the largest influence on insert operations.

6.4 Closing Remarks

In all our experiments it never happened that the main table was rehashed because of imbalances of the second level. So, to a great degree, the runtime behavior depends on the number of complete rehashes of the main table.

It is hard to give definite rules for choosing $\alpha_{\max}$ and k, especially if we do not know anything about the size of the dictionary. In general, we can say that a small $\alpha_{\max}$ ($\alpha_{\max} < 2$) is a good choice. If we handle a dictionary with few insertions and deletions and mostly lookups, then we should choose $k \in \{1, \ldots, 5\}$, but never choose $k = 0$ or $k \geq \alpha_{\max}$. For a dynamic dictionary, larger k can be taken.

The question of a good resizing factor can be answered in the following way: If one can afford to use the extra amount of memory needed by a larger resizing factor, one will get a little better runtime performance. However, note that in all our experiments we measured the average CPU time of operations. We did not care about the real time, which also includes swapping on a hard disk in case all of the main memory of the computer is in use.

ACKNOWLEDGMENTS

We thank Martin Löbbing and Martin Sauerhoff for helpful comments during the implementation and Ulrich Pfeifer for providing CPU time on the Sparc 10.

The remarks of the referees for a first version of this paper are gratefully acknowledged, as well as help from the organizers and the other participants of the Fifth DIMACS implementation challenge. We also thank a referee for the final version for the careful reading of the manuscript and the help in improving the presentation.

REFERENCES

CARTER, J. AND WEGMAN, M. 1979. Universal classes of hash functions. *J. Comput. Syst. Sci. 18*, 2 (Apr.), 143–154.

CZECH, Z. J., HAVAS, G., AND MAJEWSKI, B. S. 1992. An optimal algorithm for generating minimal perfect hash functions. *Inform. Process. Let. 43*, 5 (Oct.), 257–264.

DIETZFELBINGER, M., KARLIN, A., MEHLHORN, K., MEYER AUF DER HEIDE, F., ROHNERT, H., AND TARJAN, R. E. 1994. Dynamic perfect hashing: Upper and lower bounds. *SIAM J. Comput. 23*, 4 (Aug.), 738–761.

DIETZFELBINGER, M., HAGERUP, T., KATAJAINEN, J., AND PENTTONEN, M. 1997. A reliable randomized algorithm for the closest-pair problem. *J. Algor. 25*, 1 (Oct.), 19–51.

FREDMAN, M. L., KOMLÓS, J., AND SZEMERÉDI, E. 1984. Storing a sparse table with O(1) worst case access time. *J. ACM 31*, 3 (July), 538–544.

MCGEOCH, C. 1996a. Challenge project ideas. `http://www.cs.amherst.edu/ccm/challenge5/ideas.html`.

McGeoch, C. 1996b. The fifth dimacs implementation challenge: Data type definitions and specifications. http://www.cs.amherst.edu/ccm/challenge5/documents/specs.ps.

Thorup, M. 2000. Even strongly universal hashing is pretty fast. In *SODA '00: Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 496–497.

Received August 2002; revised August 2007; accepted January 2008