Simpler and Faster Static AC^0 Dictionaries*

Torben Hagerup

Max-Planck-Institut für Informatik, D–66123 Saarbrücken, Germany
torben@mpi-sb.mpg.de

Abstract. We consider the static dictionary problem of using $O(n)$ w-bit words to store n w-bit keys for fast retrieval on a w-bit AC^0 RAM, i.e., on a RAM with a word length of w bits whose instruction set is arbitrary, except that each instruction must be realizable through an unbounded-fanin circuit of constant depth and $w^{O(1)}$ size, and that the instruction set must be finite and independent of the keys stored. We improve the best known upper bounds for moderate values of w relative to n. If $w/\log n = (\log\log n)^{O(1)}$, query time $(\log\log\log n)^{O(1)}$ is achieved, and if additionally $w/\log n \ge (\log\log n)^{1+\epsilon}$ for some fixed $\epsilon > 0$, the query time is constant. For both of these special cases, the best previous upper bound was $O(\log\log n)$.

1 Introduction

The *static dictionary problem* is one of the most fundamental data-structuring problems. Informally, an instance of the problem is given by a set X of *keys*, each with associated *satellite data*, and the task is to store X in a way that allows rapid retrieval of the satellite data of a given key. Formally, we fix a *universe* U of possible key values and consider an instance of the problem to be given by a finite subset $X \subseteq U$, called the *key set*. A *static dictionary* for the key set X is a data structure $\mathcal{D}$ that supports *searches* for elements of U, as follows: If $x \in X$, a search for x in $\mathcal{D}$ returns $\pi(x)$, called the *index* of x, where π is an arbitrary but fixed bijection from X to $\{1, \ldots, |X|\}$; if $x \in U \setminus X$, a search for x in $\mathcal{D}$ may return an arbitrary element of $\{1, \ldots, |X|\}$. The *static dictionary problem* is to realize a static dictionary $\mathcal{D}$ for a given key set X, parameters of interest being the space occupied by $\mathcal{D}$ and the *query time*, the time needed to carry out a search in $\mathcal{D}$, but not the time needed to construct $\mathcal{D}$ from X.

Since our formal definition of the static dictionary problem is somewhat nonstandard and, at first glance, may appear rather different from the informal description, we argue that the two are, in fact, quite close. In order to obtain a static dictionary in the formal sense from one in the informal sense, we can simply store the index of each key as its associated satellite data. And to go the other way, we can support satellite data by interpreting each index as a pointer into a table of satellite data or, if the keys have different amounts of associated satellite data, as a pointer into a table that in turn contains pointers to the satellite data (see Fig. 1).

* Part of this work was carried out while the author held a visiting position at the Department of Computer Science, University of Copenhagen, Denmark.

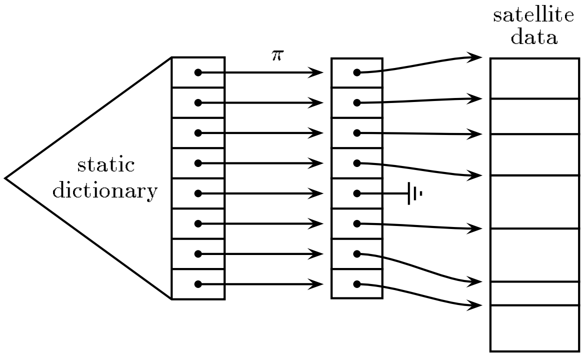

Fig. 1. A static dictionary augmented with satellite data.

The model of computation used in this paper is the *word RAM*. This model is related to the classic unit-cost RAM of Cook and Reckhow [3], the main difference being that, for a certain integer parameter $w \geq 1$ called the *word length*, all values stored in memory cells are nonnegative w-bit integers (i.e., elements of $\{0, \ldots, 2^w - 1\}$), sometimes identified with strings of w bits each and called *words*. As for the classic RAM, we assume a word RAM to have constant-time instructions for executing direct and indirect loads and stores as well as conditional and unconditional jumps. In addition, the instruction set of a word RAM contains a finite number of constant-time *arithmetic instructions*, each of which maps a constant number of operand words (usually two words) to a single result word. We always assume the arithmetic instruction set to be a superset of the *restricted instruction set*, which comprises addition and subtraction modulo 2^w, left and right shifts (with zero filling) by a variable number of bit positions, as well as the bitwise Boolean operations AND, OR, and NOT. Additional instructions will be specified in the following. Specializing the static dictionary problem to the word RAM with word length w, we fix the universe U of possible keys to be the set $\{0, \ldots, 2^w - 1\}$. We focus on *linear-space* dictionaries, ones that store n keys using $\Theta(n)$ w-bit words. We will assume that $w \geq 2 \log n$ (all logarithms in the paper are to base 2), so that we can actually address $\Theta(n)$ words of storage.

A class $\mathcal{H}$ of functions from U to a finite set S is said to be *universal* if there is a constant $c > 0$ such that for all $x, y \in U$ with $x \neq y$, $|\{h \in \mathcal{H} : h(x) = h(y)\}| \leq c|\mathcal{H}|/|S|$ (several related definitions are common). In a celebrated result, Fredman et al. [5] showed that if for each integer $s \geq 1$ there is a universal class $\mathcal{H}_s$ of functions from U to $\{0, \ldots, s-1\}$ (such that the classes share a common value of the implicit constant c), each of whose functions can be represented in a constant number of words and evaluated in constant time when s is bounded by the size of the key set, then the static dictionary problem has a linear-space solution with constant query time. Such families of universal classes actually exist. E.g., the original formulation of Fredman et al. used $\mathcal{H}_s = \{x \mapsto (kx \bmod p) \bmod s \mid 1 \leq k < p\}$, for $s = 1, 2, \ldots$, where p is an arbitrary prime larger than s. While the resulting static dictionary is a very appealing data structure, it yields constant query time only under the assumption that constant-time multiplication and integer division are available. It has been argued that this assumption is not realistic for large word lengths because the operations of multiplication and integer division are not *AC^0 operations* [6,8], i.e., they cannot be realized with unbounded-fanin circuits of constant depth and polynomial size (which, in the present context, means $w^{O(1)}$ size). Motivated by such concerns, Andersson et al. [1] studied the problem of implementing *static AC^0 dictionaries*, static dictionaries whose search operations use only AC^0 instructions. We continue this study. For reasons of possible practical relevance as well as not to trivialize the problem, the set of AC^0 instructions used is required to be finite and independent of the key set stored. All instructions in the restricted instruction set are AC^0 instructions.

Many families of universal classes of functions other than the one described above have been proposed; in particular, Dietzfelbinger et al. [4] showed how to eliminate the need for integer division and get by with multiplication as the only instruction outside the restricted instruction set. However, none of these classes consists of functions that can be evaluated in constant time using only AC^0 instructions and, indeed, Mansour et al. [9, Theorem 6.3(1)] proved that such a class cannot exist (for a slightly different notion of universality). While this does not in itself imply anything about the existence of fast static AC^0 dictionaries, it does suggest that maybe universal classes are not the way to go. It turns out, however, that a modification of a well-known universal class yields an efficient construction.

Carter and Wegman, who introduced the concept of a universal class, proved that for any two finite-dimensional vector spaces V and W over the two-element field, the class of all linear mappings from V to W is universal [2, Proposition 9]. Thus if r and b are positive integers and $\mathcal{M}^{r\times b}$ is the set of all $r \times b$ matrices with entries in $\{0,1\}$, then the class $\mathcal{H} = \{x \mapsto Ax \bmod 2 \mid A \in \mathcal{M}^{r\times b}\}$ is universal. (Recall that we identify integers with bit strings, which here in turn are identified with 0-1 column vectors in the obvious way). Premultiplication with arbitrary matrices in $\mathcal{M}^{r\times b}$ modulo 2 is not an AC^0 operation. However, Andersson et al. [1] constructed a static AC^0 dictionary based on the following two observations: (1) If A is *sparse*, i.e., if each row of A contains only a small number of entries equal to 1, then premultiplication with A may be easy (depending on the representation of A). (2) Although two keys may be very likely to collide (be mapped to the same value) under premultiplication with a randomly chosen sparse matrix, this happens only for keys whose binary representations largely coincide, and collisions between such keys are easier to handle. The static AC^0 dictionary of Andersson et al. has query time

$$O\left(\min\left\{\frac{\log w(\log\log\log n - \log\log\log w)}{\log\log w}, \sqrt{\frac{\log n}{\log\log n}}\right\}\right)$$

and uses linear or near-linear space (a linear space bound is claimed, but does not appear obvious from the (recursive) construction).

We reuse some of the building blocks of Andersson et al. [1], in particular, those underlying what is here called sampling and block compression, but implement one of them more efficiently and put them together in a simpler and cleaner way that achieves a stronger result. Taking $z = w/\log n$, we achieve a query time of

$$O\left(\min\left\{(\log z)^{\log 3/\log(3/2)}, \left(1+\frac{\log z}{\log\log w}\right)\cdot 2^{2\log z/\log(2+z/\log w)}, 1+\frac{\log n}{\log w}\right\}\right)$$

together with a linear space bound. The exponent $\log 3/\log(3/2)$ is approximately 2.71.

If we eliminate the parameter w to obtain a bound that depends only on n (thus, for every value of n, w is chosen in a worst-case manner), the bound of Andersson et al. [1] and the new bound both simplify to $O(\sqrt{\log n/\log\log n})$, which matches a lower bound of $\Omega(\sqrt{\log n/\log\log n})$ of Andersson et al. For moderate values of w, however, namely as long as $w = 2^{(\log n)^{o(1)}}$, the new bound is stronger than the bound of Andersson et al., and it is never weaker. E.g., for $z = (\log\log n)^{O(1)}$, the new query time is always $O((\log\log\log n)^{2.71})$, and if additionally $z \ge (\log\log n)^{1+\epsilon}$ for some fixed $\epsilon > 0$, the query time is constant. In both cases, the bound of Andersson et al. is $O(\log\log n)$. For this range of w of arguably greatest practical relevance—in realistic situations, w is larger than $\log n$, but not much larger—we thus achieve what is sometimes called an "exponential improvement". Just as for the data structure of Andersson et al., standard methods can be used to derive from our static dictionary a *dynamic* randomized dictionary with the same query time and deletion and expected amortized insertion bounds of the same order.

2 Overview

In this section we first cite two previous results to which we will appeal repeatedly and then give an overview over our construction.

Multiplication and integer division of b-bit integers can be carried out by looking up the result in tables of $O(2^{2b}b)$ bits, and multiplication and integer division of $O(b)$-bit integers reduce to multiplication and integer division of b-bit integers via standard algorithms for multiple-precision arithmetic. The result of Fredman et al. [5] discussed in the introduction therefore implies the following.

Lemma 1 ([1]). *For $w = O(\log n)$, there is a linear-space static AC^0 dictionary for n keys with constant query time.*

The following result, in contrast, provides a linear-space static dictionary with constant query time for sufficiently large values of w relative to n.

Lemma 2. *There is a linear-space static AC^0 dictionary for n keys with query time $O(1+\log n/\log w)$.*

Proof. Hagerup showed that there is even a linear-space *dynamic* AC^0 dictionary with the stated time bound [7, Theorem 6]. $\square$

In light of Lemma 2, we can assume without loss of generality that $w \leq n$. In particular, a pointer into a block of $O(nw)$ bits can be stored in $O(\log n)$ bits. We shall frequently need such pointers to locate various parts of a static dictionary. Since $O(\log n)$ bits will always be a negligible amount of storage, however, such pointers will not be mentioned explicitly.

Suppressing a few details, we can describe the task of a static dictionary as that of mapping a key set X injectively to the set $\{1, \ldots, |X|\}$. We solve this problem by first mapping $X = X_0$ injectively to a set X_1 of keys of fewer bits, then mapping X_1 to a set X_2 of still shorter keys, and so on, until finally X_k is mapped injectively to $\{1, \ldots, |X|\}$, for some $k \geq 0$. For $i = 1, \ldots, k$, the mapping of X_{i-1} to X_i is called a *reduction*, and each reduction will access its own auxiliary data stored as part of the complete static dictionary.

Each reduction is composed of three mappings that are applied successively to the keys under consideration, the *sampling*, the *block compression*, and the *cleanup mapping*. We next describe these three mappings in turn, assuming that we are dealing with a key set X of n b-bit keys. Let s and t be parameters that will be fixed later as squares of positive integers and take $r = 8t\lceil \log n \rceil$.

The sampling maps each b-bit key to an r-bit *signature* concatenated with a b-bit *offset*. Call a matrix with entries in $\{0, 1\}$ a *sampling pattern*. The r-bit signature of a b-bit key x is obtained as $Ax \bmod 2$, where A is an $r \times b$ sampling pattern, bit strings are identified with column vectors, and the modulo operation is applied separately to each component. In the context of a particular $r \times b$ sampling pattern A, we define a *cluster* (with respect to X and A) to be an equivalence class of the equivalence relation $\sim$ on X defined by $x \sim y \Leftrightarrow Ax \equiv Ay \pmod 2$, i.e., two keys belong to the same cluster if and only if they have the same signature. In general, the signature alone will not be enough to distinguish all keys in X, i.e., some clusters may contain more than one key. The offset serves to distinguish the keys within each cluster and simply measures the bitwise difference, modulo 2, to a fixed key in the cluster called the *representative* of the cluster. If the representative of a cluster containing a key x is the key x', the b-bit offset of x is thus computed as $x \oplus x'$, where $\oplus$ denotes the bitwise exclusive-or operation, i.e., bitwise addition modulo 2. The sampling is clearly injective.

In order to compute the offset of a given key, we need to know the representative of the cluster defined by the signature of the key. We therefore store all signatures of keys in X in a separate static dictionary—realized either recursively or according to Lemma 1—and store the representatives as their satellite data. We can also use this to replace each signature by a *signature index* of $\lceil \log n \rceil$ bits, another obviously injective mapping.

The sampling replaces the original keys by longer keys, which may seem counterproductive. The point is, however, that the offsets will have small *Hamming norms*, where the Hamming norm of a bit string x, denoted $||x||$, is the number of occurrences of a 1 in x. Informally, the reason for this is that if two keys x and y belong to the same cluster and thus fail to be distinguished by their signatures, x and y will agree on many bits, so that their *Hamming distance*, $||x \oplus y||$, will be small; in particular, the Hamming distance of each key x from the representative of its cluster, which is the Hamming norm of the offset of x, will be small. More precisely, we will ensure that the Hamming norm of each offset is bounded by $b/(st)$. The block compression capitalizes on this fact.

The block compression operates on the transformed keys consisting of signature indices and offsets. It leaves the signature indices unchanged, but replaces each offset by a *directory* concatenated with a *compressed offset*. The details are as follows (see Fig. 2): The b-bit offset is viewed as a sequence of $\lceil b/\sqrt{st}\rceil$ *blocks* of

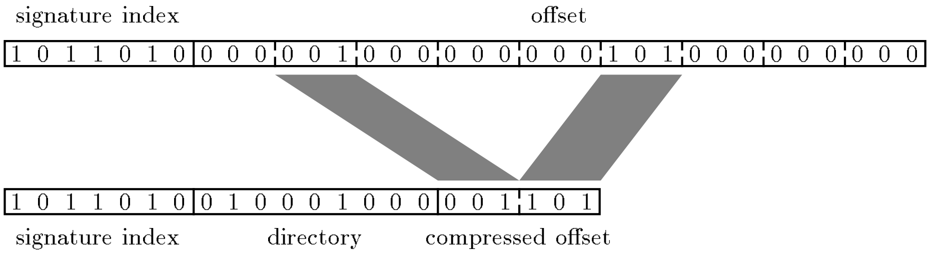

Fig. 2. The block compression.

$\sqrt{st}$ consecutive bits each, except that the last block may be smaller (recall that st is a perfect square). The directory consists of $\lceil b/\sqrt{st}\rceil$ bits, the ith of which, for $i = 1, \ldots, \lceil b/\sqrt{st}\rceil$, has the value 1 if and only if the ith block is nonzero, i.e., contains at least one bit with a value of 1. The directory thus specifies the set of nonzero blocks in a straightforward manner, and the compressed offset is simply the concatenation of the nonzero blocks in their original order. The block compression is clearly injective. Since the Hamming norm of each offset is bounded by $b/(st)$, the number of nonzero blocks is bounded by the same quantity, and thus each compressed offset consists of at most $b/(st) \cdot \sqrt{st} = b/\sqrt{st}$ bits; we append zeros as necessary to make the number of bits be exactly $\lfloor b/\sqrt{st}\rfloor$.

It turns out that we can compute a compressed offset efficiently from the corresponding offset only if we have access to certain "magic numbers" that depend on the relevant directory. For this reason we store the directories in yet another separate static dictionary—realized recursively—and store the "magic numbers" as their satellite data. As above, we can then replace each directory by a *directory index* of $\lceil \log n\rceil$ bits.

The sampling and block compression together replace each b-bit key by a signature index of $\lceil \log n\rceil$ bits, a directory index of $\lceil \log n\rceil$ bits, and a compressed offset of $\lfloor b/\sqrt{st}\rfloor$ bits, a total of $q = 2\lceil \log n\rceil + \lfloor b/\sqrt{st}\rfloor$ bits. The cleanup mapping reduces the number of bits in the keys further to $\lfloor b/\sqrt{st}\rfloor$ by replacing the leftmost $\min\{3\lceil \log n\rceil, q\}$ bits of each key by an index of $\lceil \log n\rceil$ bits by means of a static dictionary realized according to Lemma 1. The net effect of the three parts of a reduction is to reduce the number of bits in the keys from b to at most $b/\sqrt{st}$. As part of the reduction we must realize two auxiliary static dictionaries, each of which contains at most n keys. One of these dictionaries (for the signatures) contains r-bit keys, while the other (for the directories) contains keys of $\lceil b/\sqrt{st}\rceil$ bits, which we can reduce to $\lfloor b/\sqrt{st}\rfloor$ bits as in the cleanup mapping. In addition, we must deal with the transformed keys of $\lfloor b/\sqrt{st}\rfloor$ bits each. When executing a query, we must carry out one search in each of the two auxiliary dictionaries and one search for the transformed key.

In the following sections we will argue the existence of sampling patterns that cause the Hamming norms of all offsets to be sufficiently small, discuss the implementation of the sampling and block compression, realize a complete static dictionary as a cascade of successive reductions, and finally estimate the query time and the space requirements of the complete static dictionary.

3 A Reduction: The Details

For $s \geq 0$, let us call a sampling pattern A *s-sparse* if no row in A contains more than s bits with a value of 1. We will employ only s-sparse sampling patterns for relatively small values of s because such sampling patterns can be stored and applied efficiently. The following lemma is instrumental in showing the existence of suitable s-sparse sampling patterns.

Lemma 3. *Let Z be binomially distributed with parameters s and p, where s is odd. Then* $\Pr(Z \textit{ is odd}) \geq \frac{1}{2}(1 - \Pr(Z = 0))$.

Proof. Consider the Markov chain in Fig. 3, in which state 0 is the initial state and each state has a transition to itself (not shown) with probability $1 - p$, and denote by $q_i^{(k)}$ the probability of being in state i after $2k + 1$ transitions,

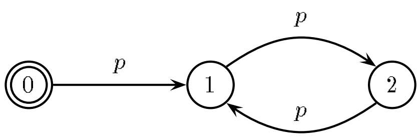

Fig. 3. A Markov chain for repeated coin tosses.

for $i = 0, 1, 2$ and $k = 0, 1, 2, \ldots$ Our task is to show that $q_1^{(k)} \geq q_2^{(k)}$, for $k = 0, 1, 2, \ldots$ For $k = 0, 1, 2, \ldots$, we have

$$q_1^{(k+1)} = 2(q_0^{(k)} + q_2^{(k)})p(1-p) + q_1^{(k)}(p^2 + (1-p)^2)$$
$$q_2^{(k+1)} = q_0^{(k)}p^2 + 2q_1^{(k)}p(1-p) + q_2^{(k)}(p^2 + (1-p)^2)$$

and hence

$$q_1^{(k+1)} - q_2^{(k+1)} = q_0^{(k)}p(2 - 3p) + (q_1^{(k)} - q_2^{(k)})(2p-1)^2.$$

If $p \le 2/3$, $q_1^{(k)} - q_2^{(k)} \ge 0$ clearly implies $q_1^{(k+1)} - q_2^{(k+1)} \ge 0$. Since $q_1^{(0)} = p$ and $q_2^{(0)} = 0$, the claim follows by induction. If $p > 2/3$, we prove by induction that for $k = 0, 1, 2, \ldots$,

$$q_1^{(k)} - q_2^{(k)} \ge p(2p-1)^{2k}(1 - u + u^{2k+1}),$$

where $u = (1-p)/(2p-1)$; note that $0 \le u \le 1$. The induction basis, for $k = 0$, follows as above. As for the inductive step from k to $k+1$, for $k \ge 0$, we use $q_0^{(k)} = (1-p)^{2k+1}$ and the induction hypothesis to conclude that

$$\begin{aligned} q_1^{(k+1)} - q_2^{(k+1)} &\ge p(2p-1)^{2k+2}(1 - u + u^{2k+1}) - (1-p)^{2k+1}p(3p-2) \\ &= p(2p-1)^{2k+2}\left(1 - u + u^{2k+1}\left(1 - \frac{3p-2}{2p-1}\right)\right) \\ &= p(2p-1)^{2k+2}(1 - u + u^{2k+2}) \\ &\ge p(2p-1)^{2(k+1)}(1 - u + u^{2(k+1)+1}). \quad \Box \end{aligned}$$

A lemma similar to Lemma 4 below was stated without proof by Andersson et al. [1, Lemma 12].

Lemma 4. *Let n, b, s and t be positive integers, where s is odd, take $r = 8t\lceil \log n \rceil$ and let X be a set of n strings of b bits each. Then there is an s-sparse $r \times b$ sampling pattern A with the property that for all $x, y \in X$ with $||x \oplus y|| \ge b/(st)$, we have $Ax \not\equiv Ay \pmod 2$.*

Proof. We use the *probabilistic method* and exhibit a random process that, with nonzero probability, yields a sampling pattern with the property mentioned in the lemma. The random process is simple: Starting from an all-zero sampling pattern, we process the r rows independently of each other. For each row, we s times in succession choose a position in the row at random from the uniform distribution over the set of all b positions and independently of all other such choices and invert the bit stored in that position.

Fix two arbitrary elements x and y of X with $||x \oplus y|| \ge b/(st)$ and consider a particular row a of A, viewed as a random quantity. Since $(ax + ay) \bmod 2 = a(x \oplus y) \bmod 2$, we will have $ax \not\equiv ay \pmod 2$ and hence $Ax \not\equiv Ay \pmod 2$ if a contains an *odd* number of bits equal to 1 in positions in which $x \oplus y$ holds a 1. The latter is the case exactly if the number Z of inversions of bits of a carried out in such positions is odd (even though the two numbers may not coincide, due to cancellations). The random variable Z is binomially distributed with parameters s and $||x \oplus y||/b \ge 1/(st)$, and we can apply Lemma 3. $\Pr(Z = 0) \le (1 - 1/(st))^s \le e^{-1/t} \le 1 - \frac{1}{2t}$, where in the last step we used the inequality $e^{-u} \le 1 - u/2$, valid for $0 \le u \le 1$. Thus $\Pr(Z \text{ is odd}) \ge \frac{1}{4t}$. In other words, a fixed row of A distinguishes between x and y with probability at least $\frac{1}{4t}$, and the probability that none of the $r = 8t\lceil \log n \rceil$ rows of A distinguishes between x and y is bounded by $(1 - \frac{1}{4t})^r \le e^{-(8/4)\lceil \log n \rceil} < 1/n^2$. The number of pairs $x, y \in X$ with $||x \oplus y|| \ge b/(st)$ is bounded by n^2, and therefore the probability that some such pair is not distinguished by A is strictly below 1. $\Box$

We next discuss the implementation of the sampling in terms of AC^0 instructions. We will represent sampling patterns in two different ways, the *bit-vector representation* and the *sparse-row representation*, which we consider separately.

The bit-vector representation. The *suitability* of a sampling pattern, its having the property mentioned in Lemma 4, is not affected by an arbitrary permutation of its rows or by the replacement of all but one occurrence of a particular row by all-zero rows. We can therefore assume without loss of generality that the rows of a sampling pattern are sorted in nonincreasing lexicographic order and that the nonzero rows are all distinct. This is useful in the case of a 1-sparse $r \times b$ sampling pattern A, which, if it is of this restricted kind, can be represented simply by a b-bit vector whose ith bit is 1, for $i = 1, \ldots, b$, exactly if the ith column of A contains a (single) 1. We will call a bit vector used in this way a *sampling vector* and speak of the bit-vector representation (see Fig. 4).

$$\begin{pmatrix}
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{pmatrix}$$

$$\begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|}
\hline
0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\
\hline
\end{array}$$

Fig. 4. A sorted 1-sparse sampling pattern and the corresponding sampling vector.

The mapping represented by a sampling vector can be visualized in a particularly simple way (see Fig. 5): The sampling vector specifies a set of bit positions

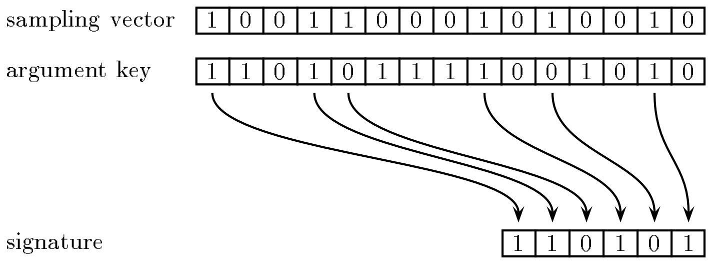

Fig. 5. The mapping represented by a sampling vector.

to be "sampled", and the bits of the argument key in these positions are extracted and concatenated in their original order. In order to realize this operation by means of AC^0 instructions, we begin by connecting each input "conditionally" to each output. More precisely, we consider a circuit that maps the b bits of a key to the r bits of its signature and in which each of the r output bits is obtained as the disjunction of b bits, each of which is the conjunction of a different input bit and a *control bit*. What is still missing is circuitry to compute values for the br control bits that precisely establish the desired connections from input bits to output bits. Let us number the input and output bits as well as the bits of the sampling vector from the right starting at 1 and denote the ith bit of the sampling vector by v_i, for $i = 1, \ldots, b$. It is easy to see that the control bit that establishes a connection from the ith input bit to the jth output bit should have the value 1 exactly if $(v_i = 1) \wedge (d_i = j)$, where $d_i = \sum_{l=1}^{i} v_l$, for $i = 1, \ldots, b$ and $j = 1, \ldots, r$. Provided that $d_1, \ldots, d_b$ are available, the control bits are very easy to compute with an AC^0 circuit. The prefix sums $d_1, \ldots, d_b$ cannot be computed from $v_1, \ldots, v_b$ with an AC^0 circuit; however, just as we store the sampling vector as part of the static dictionary, we can store also its prefix sums. One complication is that we need $\Theta(b \log b)$ bits to represent all of $d_1, \ldots, d_b$, whereas we want to get by with $O(b)$ bits—in particular, the prefix sums should fit in a single word. We can get around this complication by using the following well-known fact.

Lemma 5. *Every Boolean function of $\lceil \log(w+1) \rceil$ bits can be computed with an unbounded-fanin circuit of constant depth and $w^{O(1)}$ size.*

To see that the fact is true, simply imagine a circuit that directly reflects the truth table of the function under consideration. We use the fact as follows: We divide the bits of the sampling vector into *groups* of $\lceil \log(w+1) \rceil$ consecutive bits each, except that the rightmost group may be smaller, and record only the prefix sums of the groups. In more detail, if the number of groups is g, we store as part of the static dictionary a sequence of $g-1$ *global prefix sums* of $\lceil \log(w+1) \rceil$ bits each, where the ith global prefix sum is the total number of bits equal to 1 in the i rightmost groups, for $i = 1, \ldots, g-1$. The number of bits needed to store the sequence of global prefix sums is clearly bounded by b, and each of the b original prefix sums can be obtained as the sum of a global prefix sum and a *local prefix sum* computed with respect to a single group. By Lemma 5, the local prefix sums can be computed from the sampling vector with an AC^0 circuit, and the addition of global and local prefix sums is also an AC^0 operation. We have thus shown how to execute the sampling mapping with a single AC^0 instruction, the *sampling instruction*, that takes as arguments a key, the sampling vector, and the sequence of global prefix sums. The storage required is at most $2b$ bits.

The sparse-row representation. In the sparse-row representation, also used by Andersson et al. [1], we store an s-sparse sampling pattern A as a sequence of r groups of s integers of $m = \lceil \log(w+1) \rceil$ bits each. For $i = 1, \ldots, r$, the s integers in the ith group specify the at most s positions in the ith row of A that contain a 1, a special bit pattern reserved to denote "no position" (in case the row contains fewer than s occurrences of a 1). We will later ensure that $rsm = O(w)$, so that the entire sampling pattern fits in a constant number of words, and we analyze the time needed to apply the sampling pattern under this assumption.

The application of the sampling pattern can be decomposed into two subtasks: Extracting the at most rs bits specified in the sampling pattern from the argument key and storing them in consecutive positions of a word (they will fit, except for n bounded by a constant); and forming the sum, modulo 2, of the bits within each of the r groups of s consecutive bits. The first subtask can be carried out in constant time using a "multiselect" instruction that is easy to devise and shown explicitly in [7, Fig. 11]. If $s = m^k$ for some integer $k \geq 1$, the second subtask can be carried out by k successive applications of a one-argument operation that views its argument as composed of segments of m bits each and replaces the bits in each segment by their sum, modulo 2 (storing the resulting bits in consecutive positions); by Lemma 5, this operation can be realized via an AC^0 instruction. If s is not a power of m, it is necessary first to reduce the number of bits within each group to the nearest smaller power of m. For each fixed value of s, this can be done with an AC^0 circuit C_s, much as above, and the circuits $C_1, \ldots, C_w$ can be combined into a single circuit that takes s as a second argument and uses the value of s to select the output of the correct circuit C_s. The combined circuit still realizes an AC^0 operation. Summing up, we can execute the sampling using the sparse-row representation in $O(1 + \log s / \log m) = O(1 + \log s / \log\log w)$ time.

Having dealt with the sampling, we turn to the block compression. First, the directory is very easy to compute with an AC^0 instruction. As for the computation of the compressed offset, it is similar to the sampling according to a sampling vector and can be carried out in constant time in the same way, provided that a suitable b-bit "sampling vector" and its global prefix sums are available as satellite data of the relevant directory (these are the "magic numbers" alluded to earlier). The "sampling vector" should have a 1 precisely in each position belonging to a nonempty block.

4 The Main Result

Theorem 6. *For all integers $n \geq 4$ and $w \geq 2\log n$ and for all sets X of n w-bit integers, there is a static dictionary for the key set X that works on a word RAM with a word length of w bits and a finite and fixed instruction set containing only AC^0 instructions, uses $O(n)$ w-bit words of storage, and has query time*

$$O\left(\min\left\{(\log z)^{\log 3/\log(3/2)}, \left(1 + \frac{\log z}{\log\log w}\right) \cdot 2^{2\log z/\log(2+z/\log w)}, 1 + \frac{\log n}{\log w}\right\}\right),$$

where $z = w/\log n$.

Proof. We consider the three parts of the bound one by one. In order to show the validity of the first bound, $O((\log z)^{\log 3/\log(3/2)})$, we take $s = 1$ and represent all sampling patterns according to the bit-vector representation. For a reduction that inputs keys of $b \geq \log n$ bits each, we choose t as the square of a positive integer such that $t = \Theta((b/\log n)^{2/3})$. The reduction spawns three new instances of the static dictionary problem (for brevity: *instances*), and the value of t was chosen to make all three instances involve keys of $O(b^{2/3}(\log n)^{1/3})$ bits each. For a suitable constant $c > 0$, this implies that the derived quantity $\log(cb/\log n)$ is reduced by a factor of at least $3/2$ from each level of recursion to the next. Since the derived quantity starts out at $O(\log(w/\log n)) = O(\log z)$ and we can end the recursion according to Lemma 1 when it reaches 1, the depth of recursion will be $\log\log z/\log(3/2) + O(1)$. But then both the total number of instances spawned and the query time will be $O(3^{\log\log z/\log(3/2)}) = O((\log z)^{\log 3/\log(3/2)})$.

A reduction that inputs keys of $b \geq \log n$ bits needs $O(nb)$ bits of storage. Each new level of recursion triples the number of instances, but except for a constant number of levels just before the recursion bottoms out, each recursive level reduces the number of bits per key by a factor of more than 6. This can be seen to imply that the total space requirements are $O(nw)$ bits or $O(n)$ words of w bits each.

In order to show the validity of the second bound of Theorem 6, observe first that we can assume that $z/\log w \geq 16$, since otherwise the first bound is surely no larger than the second bound. We take $t = 1$, choose s as a square of an odd integer with $s \geq 2 + z/\log w$, but $s = O(z/\log w)$, and represent all sampling patterns according to the sparse-row representation. The condition $rsm = O(w)$, imposed in the discussion of the sparse-row representation, now takes the form $\lceil\log n\rceil\lceil\log(w+1)\rceil s = O(w)$ and is easily seen to be satisfied, so that each reduction can be executed in $O(1 + \log z/\log\log w)$ time. The choice $t = 1$ implies that the static dictionaries storing signatures can be implemented directly using Lemma 1, for which reason each reduction now spawns only two new instances. Each recursive level reduces the number of bits in the keys under consideration by a factor of at least $\sqrt{s}$, so that the required depth of recursion is $\log z/\log\sqrt{s} + O(1) = 2\log z/\log(2 + z/\log w) + O(1)$. The total number of instances spawned is thus $O(2^{2\log z/\log(2+z/\log w)})$, and the query time is $O((1 + \log z/\log\log w) \cdot 2^{2\log z/\log(2+z/\log w)})$, as claimed.

A space bound of $O(nw)$ bits follows essentially as in the case of the first bound, noting that $\sqrt{s} \geq 4$. One difference is that each reduction now needs $O(w)$ bits to store its sampling pattern. Since we can assume that $w \leq n$, the additional space requirements are negligible. The third bound of Theorem 6, finally, is just a restatement of Lemma 2. $\square$

References

1. A. Andersson, P. B. Miltersen, S. Riis, and M. Thorup, Static dictionaries on AC^0 RAMs: Query time $\Theta(\sqrt{\log n/\log\log n})$ is necessary and sufficient, in Proc. 37th Annual IEEE Symposium on Foundations of Computer Science (FOCS 1996), pp. 441–450.
2. J. L. Carter and M. N. Wegman, Universal classes of hash functions, *J. Comput. System Sci.* **18** (1979), pp. 143–154.

3. S. A. Cook and R. A. Reckhow, Time bounded random access machines, *J. Comput. System Sci.* **7** (1973), pp. 354–375.
4. M. Dietzfelbinger, T. Hagerup, J. Katajainen, and M. Penttonen, A reliable randomized algorithm for the closest-pair problem, *J. Algorithms* **25** (1997), pp. 19–51.
5. M. L. Fredman, J. Komlós, and E. Szemerédi, Storing a sparse table with $O(1)$ worst case access time, *J. Assoc. Comput. Mach.* **31** (1984), pp. 538–544.
6. M. Furst, J. B. Saxe, and M. Sipser, Parity, circuits, and the polynomial-time hierarchy, *Math. Syst. Theory* **17** (1984), pp. 13–27.
7. T. Hagerup, Sorting and searching on the word RAM, in Proc. 15th Symposium on Theoretical Aspects of Computer Science (STACS 1998), Lecture Notes in Computer Science, Springer, Berlin.
8. J. Hastad, Almost optimal lower bounds for small depth circuits, in Proc. 18th Annual ACM Symposium on Theory of Computing (STOC 1986), pp. 6–20.
9. Y. Mansour, N. Nisan, and P. Tiwari, The computational complexity of universal hashing, *Theoret. Comput. Sci.* **107** (1993), pp. 121–133.