Improved Address-Calculation Coding of Integer Arrays*

Amr Elmasry[1,2], Jyrki Katajainen[1], and Jukka Teuhola[3]

[1] University of Copenhagen, Department of Computer Science, Denmark
[2] Alexandria University, Computer and Systems Engineering Department, Egypt
[3] University of Turku, Department of Information Technology, Finland

Abstract. In this paper we deal with compressed integer arrays that are equipped with fast random access. Our treatment improves over an earlier approach that used address-calculation coding to locate the elements and supported access and search operations in $O(\lg(n+s))$ time for a sequence of n non-negative integers summing up to s. The idea is to complement the address-calculation method with index structures that considerably decrease access times and also enable updates. For all our structures the memory usage is $n \lg(1 + s/n) + O(n)$ bits. First a read-only version is introduced that supports rank-based accesses to elements and retrievals of prefix sums in $O(\lg\lg(n+s))$ time, as well as prefix-sum searches in $O(\lg n + \lg\lg s)$ time, using the word RAM as the model of computation. The second version of the data structure supports accesses in $O(\lg\lg U)$ time and changes of element values in $O(\lg^2 U)$ time, where U is the universe size. Both versions performed quite well in practical experiments. A third extension to dynamic arrays is also described, supporting accesses and prefix-sum searches in $O(\lg n + \lg\lg U)$ time, and insertions and deletions in $O(\lg^2 U)$ time.

1 Introduction

Compressed data structures, such as arrays and dictionaries, have received much attention lately (see, for example, [2, 4, 9, 11–13, 16–20]). In addition to conciseness, an essential property is the support of efficient operations directly on the compressed form. The underlying data collections can be sequences, sets, or multisets—either sorted or unsorted—of numeric or symbolic elements.

Using binary coding, an integer x can be represented in $\lceil \lg(1 + x) \rceil$ bits[1], assuming that the length is known. By the same token, a *raw representation* of a *sequence* of non-negative integers $x_1, x_2, \ldots, x_n$ requires $\sum_{i=1}^{n} \lceil \lg(1 + x_i) \rceil$ bits. Gupta et al. [9] called this type of measures *data-aware*. By denoting $s = \sum_{i=1}^{n} x_i$ and using Jensen's inequality, the length of the raw representation is seen to be bounded by $n \lg(1 + s/n) + n$. This bound is roughly the same as $\left\lceil \lg \binom{s-1}{n-1} \right\rceil$, which is the minimum number of bits required to store any sequence of n positive integers that add up to s [5]. Later on we assume that the numbers manipulated

[1] Throughout the paper we use $\lg x$ as a shorthand for $\log_2(\max\{2, x\})$.

are all smaller than U, which is an upper bound on any number that fits in a machine word. Therefore, $n \lg(U/n)$ is yet another bound that is independent of the original values.

Any of the above bounds could be used as the basis of defining compactness of a coding system. Since we want to decode the numbers uniquely and provide random access to them, in addition to the raw representation, extra memory usage is unavoidable. The *overhead* of a code indicates how close the actual memory consumption is to one such bound. In this paper we consider the problem of storing an array of integers such that the length of the representation is close to the information-theoretic optimum. In particular, we say that an array storing non-negative integers is *compact* if the number of bits used by its representation is at most $n \lg(1 + s/n) + O(n)$. For a uniform distribution, a compact code has an overhead of $O(n)$ compared to the entropy.

The mandatory operation for any array, in addition to initial construction, is access by *rank*, often called array index. For sorted sequences, search by element value is quite often useful as well. The corresponding operation for unsorted sequences is search by prefix sum, and conversely accessing the prefix sum by index. As for updates, arrays can be classified into three types:

1. read-only, called here *static arrays*,
2. fixed-length updatable, called here *modifiable arrays*, and
3. variable-length mutable, called here *dynamic arrays*.

The interesting operations on arrays, depending on the array type, include:

- retrieve the ith element, denoted here *access*(i),
- retrieve the sum of elements $1, \ldots, i$, called also *prefix sum*, denoted *sum*(i),
- find the rank of a given prefix sum p, denoted *search*(p),
- change the ith element value to v, denoted *modify*(i, v),
- insert a new element v before position with index i, denoted *insert*(i, v),
- remove the ith element, denoted *delete*(i).

The common application of compressed arrays is information retrieval using *inverted lists* (see, e.g. [4, 14, 15]). The intersection of these lists can be calculated faster if one of the lists is short and its elements are searched from the longer directly. With three or more keywords the advantage becomes even bigger, because the intermediate intersections get shorter. Another application is mining of *frequent itemsets* and *association rules* from large data matrices [1]. Calculation of the support value of an itemset requires intersection of item columns. The matrices are typically sparse, so compressed arrays are an interesting option. Again, the advantage is bigger for a larger number of intersected item types.

A number of solutions have been proposed for the task of representing and accessing compressed collections. Theoretical approaches usually aim at *succinct data structures*, which have low overhead and still provide fast operations in the compressed form. Static data-aware arrays, for which the access time is constant and the overhead superlinear in n, were presented in [7, 8, 11, 18]. Jansson et al. [11] can also handle the modify operation efficiently. Raman et al. [17] get

remarkably close to the optimal data-independent bounds in compression of arrays, sets and multisets, still supporting constant-time access. Gupta et al. [9] provide efficient data structures for representing sorted sets in a data-aware manner under various operations. Practical approaches emphasize on a simple implementation and effectiveness in the contemporary processor and memory architectures. Examples include the works of Culpepper and Moffat [4], Brisaboa et al. [2], Transier and Sanders [20], and Teuhola [19].

We restrict ourselves to compact representations of integer arrays. Our approach is primarily practical and is based on a recent address-calculation coding of static arrays [19], which is reviewed in Section 2. Strikingly, our method improves the access time from $O(\lg(n+s))$ to $O(\lg\lg(n+s))$. Also, we can support the modify operation in $O(\lg^2 U)$ time, where U is the universe size. For this version, the access time is $O(\lg\lg U)$. These improvements are presented and analysed in Section 3, where we also present a way of making the array dynamic, supporting insertions and deletions but at the cost of increasing the access time to $O(\lg n+\lg\lg U)$. Experimental results for both static and modifiable arrays are reported in Section 4. The results confirm the effectiveness of the new compact array realization. The paper ends with a discussion in Section 5.

Our model of computation is the word RAM [10], augmented with an instruction that can be used to determine the most-significant one-bit of a word. Current processors usually provide such an instruction; and if not, this operation can be accomplished in constant time utilizing word-level parallelism and some precomputed tables [3]. The word length is denoted by $\lceil \lg U \rceil$, and the (normal) tacit assumption is that the lengths of the variable values do not exceed this length. In practice, $\lg U$ is of the same magnitude as $\lg n$ and $\lg s$ when comparing complexities.

2 Review of Address-Calculation Coding

The remarkable feature of address-calculation coding (*AC coding* for short), described by Teuhola [19], is that it enables relatively fast (log-time) operations on a sequence of integers without any explicit index structure or other auxiliary data, and yet uses locally adaptive variable-length coding. The starting point was *interpolative coding*, suggested by Moffat and Stuiver [14] for sorted sequences with inverted indexes in mind. Another view of the method is obtained by considering the successive *gaps* in the sequence as basic elements, and building a complete binary tree of pairwise sums with the elements as leaves. Each internal node is the sum of its children, and the root has the total sum.

In addition to the root only the left children need to be encoded, because the right ones are obtained by subtraction. Altogether the tree will contain $2n-1$ nodes, n of which need to be encoded. It is self-evident that the tree can be stored implicitly in depth-first order. The root of the tree is encoded using some universal code. The encoding of other nodes is based on the knowledge that the node value is between 0 and the parent value. Thus, fixed-length binary coding, truncated to the code length of the parent value, can be used.

The contribution of AC coding was to derive a rather tight upper bound $B(n, s)$ on the number of bits required by a sequence of n non-negative integers having a known sum s. Signed integers could be incorporated by a simple mapping to even/odd positive integers. We repeat the formula from [19]:

$$B(n,s) = \begin{cases} n(t - \lg n + 1) + \lfloor \frac{s(n-1)}{2^{t-1}} \rfloor - t - 1 & \text{, if } s \geq n/2 \\ 2^t + \lfloor s(2 - \frac{1}{2^{t-1}}) \rfloor - t - 1 + s(\lg n - t) & \text{, otherwise} \end{cases} \quad (1)$$

where $t = \lceil \lg(1 + s) \rceil$. In the formula it is assumed that n is a power of two, but this restriction is easy to handle. By using formula (1), we can reserve space first for the whole tree then (after determining the left child) for the left subtree, implying the starting address of the right subtree, and continuing recursively. These addresses to bits enable normal tree-traversal operations, because on each path we only need to decode the left child of the current node to proceed one step down. A crucial property of formula (1) should be emphasized: It holds recursively, so that the space allocated to the subtrees is never more than the space reserved for the whole tree minus the root. For this reason, the formula cannot take advantage of truncation in coding, but assumes always $\lceil \lg(1 + m) \rceil$ bits for an integer in $0 \,..\, m$. This is also one reason for the occurrence of some unused bits here and there.

The supported operations in [19] are: access(i), sum(i) and search(p), all in logarithmic time. The bound on the bit count of the structure is fully determined by n and s, and is at most $n \lg(1 + s/n) + O(n)$ when $s \geq n/2$. The constant factor of the linear term is about 3, but only part of it represents overhead. For example, if $s = n/2$, formula (1) gives a code length of about $2n$ bits. For sparse arrays ($s < n/2$), the bit count is even less. In the next section we use the described AC code as the core method, and build some auxiliary structures to improve its access speed and extend it with updates.

3 Extensions and Improvements

When encoding the tree of sums most bits are consumed on the lowest levels. However, on the path from the root to a leaf the step costs are equal. By cutting off the top part of the tree, we are left with a forest of small trees storing the bulk of the bits but having much faster access paths. In accordance, we need a mechanism—an *index*—for locating the correct root in the forest.

3.1 Indexed static arrays

An array of n elements with sum s is partitioned into $\lceil n/\lfloor c \cdot \lg(n + s) \rfloor \rceil$ subsequences, called *chunks*, having $\lfloor c \cdot \lg(n + s) \rfloor$ elements each (except possibly the last chunk that may be smaller). Here, c is a positive constant to be used for tuning between speed and overhead.

The chunks are stored as separate but contiguous AC-coded binary sequences. An additional index of $\lceil n/\lfloor c \cdot \lg(n + s) \rfloor \rceil$ entries is created, storing the starting

addresses of the chunks (i.e. roots of the implicit trees). The index entries are of equal width, so constant-time access to a given chunk address is easily supported. Each chunk root contains the *cumulative* sum of the previous chunks plus the current one. Knowing the total sum s, the roots are encoded with $\lceil \lg(1+s) \rceil$ bits each. The actual chunk sum is obtained by subtraction in constant time.

For clarity, we call the original AC code the *basic AC code*, and our extended version the *indexed AC code*.

Lemma 1. *Indexed AC-coded static arrays are compact.*

Proof. Pursuant to formula (1), the basic AC code is compact. Let k denote the number of elements per chunk and s_i the sum of the elements in the ith chuck. In all, the size of the chunks is at most $\sum_{i=1}^{\lceil n/k \rceil} k \lg(1+s_i/k) + O(n)$ bits. By Jensen's inequality, this is bounded by $n \lg(1+s/n) + O(n)$. Next we need to study the extra structures, namely the index and chunk roots. The number of bits consumed by the roots is $\lceil n/\lfloor c \lg(n+s) \rfloor \rceil \cdot \lceil \lg(1+s) \rceil \leq n/c + O(1)$. The index entries should be able to address at most $n \lg(1+s/n) + O(n)$ bits, because the chunks together cannot exceed the limit given in formula (1). Thus, the address size is $\lg(n \lg(1+s/n) + O(n)) = \lg n + \lg\lg(1+s/n) + O(1)$. The number of bits required by the whole chunk index is $\lceil n/\lfloor c \lg(n+s) \rfloor \rceil \cdot (\lg n + \lg\lg(1+s/n) + O(1)) \leq n/c + o(n/c)$. The sum of the sizes of the index, roots, and chunks is $O(n)$ bits, i.e. the structure is still compact. □

Using the described extensions to the basic AC-coded arrays, we now study the implementation of access(i), sum(i) and search(p).

Lemma 2. *Accessing the ith element and its prefix sum from a static array can be performed in $O(\lg\lg(n+s))$ time using indexed AC code.*

Proof. The correct chunk address can be determined in constant time from the index entry $\lfloor i/\lfloor c \cdot \lg(n+s) \rfloor \rfloor$. The same holds for its preceding chunk (if any), and the chunk sum is obtained as the difference of the two cumulative sums stored at chunk roots. The correct element within the chunk is found by applying the same log-time technique as with the basic AC code. Since the chunk contains $\lfloor c \cdot \lg(n+s) \rfloor$ elements, the correct leaf node is found in $O(\lg\lg(n+s))$ time.

The same holds for the prefix sum: It is initialized by the root of the left neighbour chunk. When walking down the chunk tree, at every step to the right child, the left child (containing the sum of the left subtree) is accumulated to the prefix sum. Thus, the time complexity is the same as for element access. □

Lemma 3. *Searching for a given (or nearest) prefix sum of a static array can be performed in $O(\lg n + \lg\lg s)$ time using indexed AC code.*

Proof. Searching presumes a different way of locating the correct chunk. The chunk roots contain the cumulative sums, so they are in ascending order. Since the index provides constant-time access to the roots, binary search can be applied to pick up the correct root. Searching continues then as in the basic AC structure, and the combined complexity is $O(\lg n + \lg\lg s)$. □

3.2 Indexed modifiable arrays

Changing the values of the elements is possible with the above structure, but much too laborious if the chunk size changes. For this reason, we present another version that is effective for both accessing and changing the ith element value.

As mentioned above, the AC code involves a certain amount of slack in space allocation, so that not nearly all value changes cause a change of chunk size. In the next version of our data structure, we introduce a bit more slack by aligning the chunks at word boundaries. The chunk sizes are thus multiples of $w = \lceil \lg U \rceil$. Modifications of elements affect the value of s, so using it as a structure parameter (e.g. chunk size $\lfloor c \lg(n+s) \rfloor$) would be problematic. It is also a question of a trade-off between overhead and update speed; our choice favours the latter. The change of memory unit is reflected in the chunk index as well. It now contains $\lceil n/(cw) \rceil$ entries of w bits each, pointing to chunks of cw elements. The chunk roots are now plain sums of the chunk elements. Maintenance of cumulative sums would be possible in logarithmic time (see [6]), but it would slow down direct access.

In order to restrict the amount of data to be relocated at updates, we adopt the so-called *zone technique*, coupled with a *rotation* operator, to organize the set of chunks (see e.g. [12]). More precisely, the chunks are rearranged into ascending order of size (measured in words); so that, within a *zone*, all chunks take an equal amount of memory. The zones are numbered according to this common size. The original chunk number is stored with the chunk and acts as a *back-pointer* to the index. Division into zones has the advantage that any two chunks within a zone number z can be swapped in $O(z)$ time.

Since each chunk (except possibly one) has cw elements, and each element is at most w bits wide (relying on the word-RAM model), the number of possible chunk sizes measured in words—and the number of zones—is at most cw. Within a zone the last chunk can be split (due to updates), so that its front part is at the end and tail at the beginning of the zone. In the context of modify(i, v), the affected chunk is *rebuilt* (in $O(\lg U)$ time). If its size changes, say from m to m' words, we delete the old chunk by moving m words from one or two chunks at the zone end to the place of the deleted chunk, creating a gap at the end. The gap is filled by moving zones $m+1\,..\,cw$ to the left by m words. Then, we make room for the updated chunk by moving zones $m'+1\,..\,cw$ by m' words to the right. The moves are realized as *rotations* of the required amount of words from front to rear, or vice versa, within the zone. Positioning the modified chunk to its new zone requires similar local adjustment as its deletion.

At most one chunk root per zone needs to be shifted in a rotation step, and its address in the index must be updated. Of course, the index update concerns also the changed chunk. Back-pointers enable this in constant time per relocated chunk. The back-pointers require $\lceil \lg(n/(cw)) \rceil$ bits each.

A small directory is needed for zone start addresses and zone rotation information, but their storage requirements are negligible.

Lemma 4. *The modifiable array using indexed AC code is compact.*

Proof. The chunk index has $O(n/w)$ entries each of size w bits, so together they constitute $O(n)$ bits. The chunk roots consume the same amount of memory. The back-pointers need less, namely $O(\lg(n/w))$ bits per pointer. The chunks themselves use the same amount of bits as in the static version, but word alignment adds to it at most $w - 1$ bits per chunk, which is less than one bit per element. The zone directory is clearly sublinear because it needs two entries for each non-empty zone. All in all the overhead remains linear. □

Lemma 5. *Accessing the ith element of a modifiable array can be performed in* $O(\lg \lg U)$ *time using indexed AC code.*

Proof. The access procedure is identical to that of the static array; only the locations of chunks can be different, and in a special case the chunk can be rotated into two pieces to the front and rear of a zone. Locating the correct chunk is still a constant-time operation. The chunk size is different, namely cw elements; so, the root-to-leaf path consists of $O(\lg \lg U)$ links. □

Lemma 6. *Changing the value of the ith element of a modifiable array can be performed in* $O(\lg^2 U)$ *time using indexed AC code.*

Proof. Rebuilding the affected chunk can be done in $O(\lg U)$ time. If the chunk size changes, its rebuilding takes $O(\lg U)$ time, and relocation requires at most two rotations per zone. Since there are $O(\lg U)$ zones and one rotation step moves $O(\lg U)$ words, the reorganization of zones takes $O(\lg^2 U)$ time. At most two chunk roots are relocated per zone. Using the back-pointers stored in the chunk heads, the index update takes $O(1)$ time per zone, and $O(\lg U)$ time in total. The overall time is dominated by rotations, and is as claimed. □

3.3 Indexed dynamic arrays

The chunk size (number of elements per chunk) was fixed in the aforementioned implementations for static and modifiable arrays. When extending the operation set by insert and delete, we allow the chunk size to vary between $cw/2$ and $2cw$ (except possibly one smaller chunk at the end).

The array-type chunk index is replaced by a balanced binary tree having between $n/(2cw)$ and $2n/(cw)$ leaves, and implemented using both child and parent pointers. The leaves contain chunk sizes and pointers to chunk roots, and the chunks contain pointers back to the leaves. Each internal tree node contains the sum of the sizes in its two children, representing the combined element count of the subtree chunks. These counts can be used as a basis for both navigating and rebalancing the tree. We call this third extension as *tree-indexed AC code*.

The chunk roots are encoded with w bits. The zones are used as in the modifiable case, storing chunks sorted according to their sizes (in words) and performing rotations to reserve and release space. The index tree nodes reserve the first zone to keep the coded sequence contiguous, but this zone is not rotated. The chunk zones are rotated when the tree zone ahead is resized.

Lemma 7. *The dynamic array using the tree-indexed AC code is compact.*

Proof. The tree index has $O(n/w)$ nodes, each using $4w$ bits, so the whole index consumes $O(n)$ bits. The chunk roots and back-pointers are $O(w)$ bits each and their count is $O(n/w)$, so altogether they consume $O(n)$ bits. As before, the word alignment of chunks still adds another $O(n)$ to the overhead. It follows that the overall overhead is linear. □

The amount of overhead can be tuned with the constant factor c. The usage of the structure becomes more explicit when analysing the operations.

Lemma 8. *Accessing the ith element of a dynamic array can be performed in $O(\lg n + \lg\lg U)$ time using tree-indexed AC code.*

Proof. We navigate the index tree down, using subtree sizes, in $O(\lg n)$ time. We then get the correct chunk head in constant time and the correct element within it in $O(\lg\lg U)$ time. The combined time complexity is $O(\lg n + \lg\lg U)$. □

Lemma 9. *Changing the ith element of a dynamic array can be performed in $O(\lg^2 U)$ time using tree-indexed AC code.*

Proof. This operation is analogous to the corresponding operation for the modifiable array, except for the way of locating the correct chunk which in this case requires $O(\lg n)$ time. Rotations of zones account for the rest of the cost. □

Lemma 10. *Inserting a new element after the ith element into a dynamic array and deleting the ith element from a dynamic array can be performed in $O(\lg^2 U)$ time using tree-indexed AC code.*

Proof. Starting from the index root, in an insertion we add one to the sizes in the nodes on the path to the correct chunk (containing the earlier ith element). If the chunk does not overflow, we proceed as in value modification. If the chunk overflows (size $> 2cw$), we split it into two, add the related two new leaves into the index and rebalance the tree if needed. Rebalancing involves $O(\lg n)$ constant-time node rotations, and the same number of node values (number of subtree elements) may have to be changed. Chunk zones are rotated to make room for the two new index nodes. A constant number of chunks are affected. So, including chunk rebuildings and zone rotations, the time complexity is $O(\lg^2 U)$.

Opposite to insertions, in deletions chunk underflow (size $< cw/2$) is possible. This is handled either by balancing the element counts with the twin chunk or joining them together. Node updates (subtract one from the size at each node on the correct root-to-leaf path) and rotations of the index tree due to rebalancing take $O(\lg n)$ time. As before, rebuilding one or two chunks plus zone rotations require $O(\lg^2 U)$ time. In total, the time complexity is $O(\lg^2 U)$. □

We finally note that the support of prefix-sum access and search operations can be realized in $O(\lg n + \lg\lg U)$ time, by including the sums of descendant element values in each node of the index tree in addition to element counts. Hereafter the cumulative sums need not be stored at chunk roots any more. Otherwise the implementations of these operations closely correspond to those in the basic AC-coded structure. The overhead is still $O(n)$ bits, i.e. the data structure is compact.

4 Experiments

We tested the efficiency of the basic AC-coded static arrays against two indexed arrays, namely static and modifiable versions. As a simplification, we used chunk sizes that are powers of two; this does not affect the complexity nor compactness results. The last chunk was padded with zeros if needed. The tests were performed on a double-processor 1.8 GHz Intel Xeon with word length $w = 32$. The timing results are averages of a sufficiently large number of repetitions to get reliable values in each case. The programs[2] were written in pure C with no special optimization other than using compilation switch `-O3`. The source data was artificially generated with no dependencies between elements.

Memory usage was, of course, measured for all structures, and compared to the entropy. We generated two kinds of pseudo sequences, namely (a) uniformly- and (b) exponentially-distributed integers. For (a), different range sizes from 2 to 1024 were tested; and for (b), powers of 2 between 1/64 and 8 were used for the parameter λ in the distribution $F(x;\lambda) = 1 - e^{-\lambda x}$. The tuning parameter c of the enhanced method was set to 4 in these tests (unless otherwise stated) so that the chunk size was $4\lg(n+s)$ elements (or $4w$ elements for the modifiable version), rounded to the nearest power of two. The compression results are shown in Fig. 1. The element count n was 1 000 000 in each case.

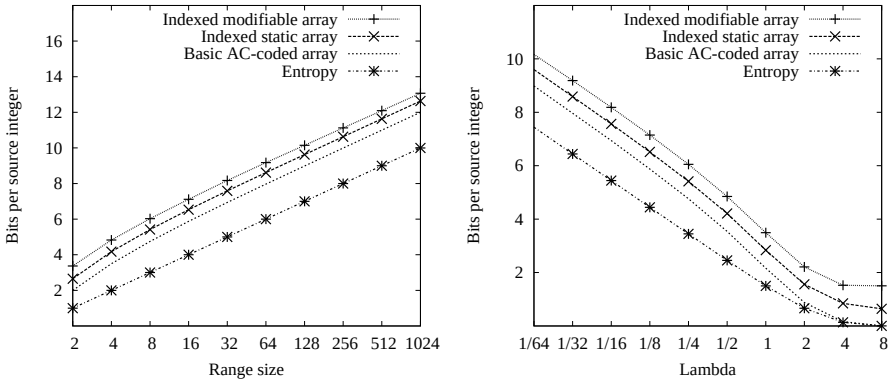

Fig. 1. Coding efficiency of the basic structure, its two extensions and the entropy, for uniformly (left) and exponentially (right) distributed integers

It can be observed that the indexed static array uses less than one bit per element more than the basic AC code, and about 2-3 bits more than the entropy. The indexed modifiable array uses still 0.5 to 1 bits more. Anyway, compactness of both structures is evident from the graphs.

Fig. 2 shows an empirical analysis of the effects of varying the parameter c that determines the chunk size and count. Increasing the value produces the

[2] Available at http://staff.cs.utu.fi/staff/teuhola/research/indexed-AC-coding.zip

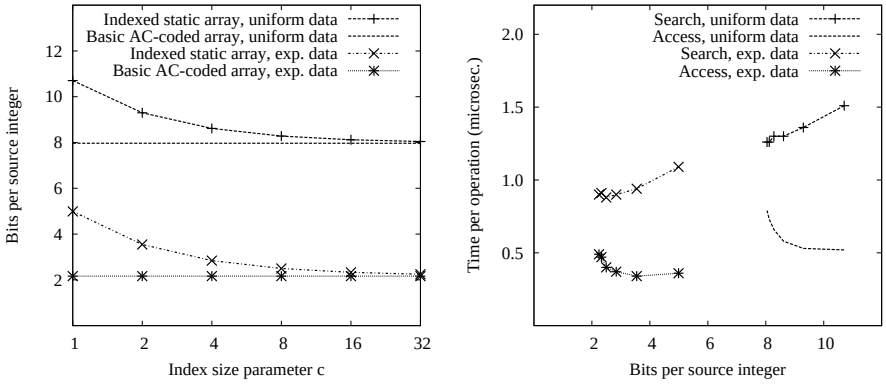

Fig. 2. Coding efficiency as a function of c (left), and coding efficiency vs. access time by varying c (right)

expected result: The code size approaches that of the basic AC code. The same figure shows also a plot of coding efficiency versus retrieval time, including both access(i) and search(v). The arrays contained $n = 1\,000\,000$ elements. Uniformly distributed elements were within $0 \ldots 63$, and $\lambda = 1$ was set for the exponential data. The curves were obtained by varying the tuning parameter c from 1 to 32 in powers of two. Searching is slower, as it should be, due to higher complexity; $O(\lg(n+s))$ vs. $O(\lg\lg(n+s))$. Interestingly, the two operations have opposite slopes. The probable reason is that binary search becomes faster for shorter index sizes. Accesses within chunks are more local than for binary search, and in accordance the number of cache misses incurred is reduced.

Time measurements for the basic AC-coded arrays and our indexed versions are shown in Fig. 3. The curves on the left show results for static arrays, and those on the right for modifiable arrays. The data set was 1 000 000 integers, uniformly distributed within $0 \ldots 63$. The access time of the ith element was the same for static and modifiable arrays, as expected. The curves indicate that the access time for our indexed AC coding of a static array is superior to that of the basic AC coding. This behaviour is expected and illustrates our main theoretical improvement of the access time from $O(\lg(n+s))$ to $O(\lg\lg(n+s))$.

For a modifiable array, changing the value of an element was about 10 times slower than a plain access. The sudden rise in the curve of modification times is due to the threshold where the chunk size is doubled. The curve represents a pessimistic case since the affected chunk is decomposed and rebuilt on each modification. In many cases this could be avoided, because there is some slack in the structure. Expectedly, the observed update time is somewhat shorter.

The conclusion is that, in addition to theoretical interest, the suggested scheme has practical utility for large sequences of numbers with skewed distributions, requiring mostly searching for element values, while offering a rank-based random-access capability as well.

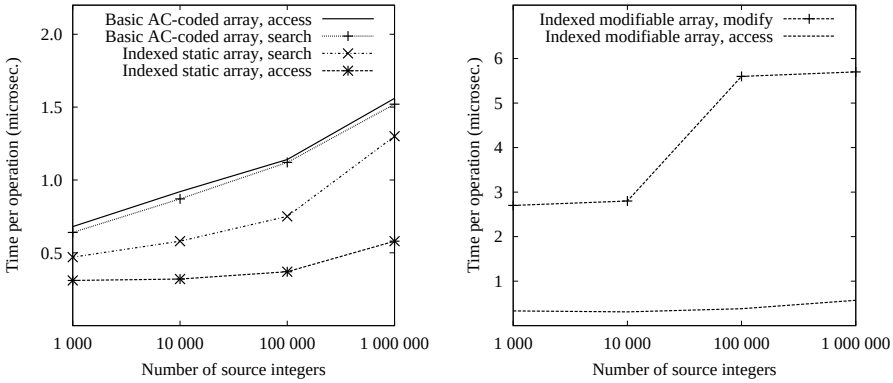

Fig. 3. Access and search times for the basic and indexed static arrays (left), and access and update times for indexed modifiable arrays (right)

5 Conclusions

We described compact data structures for representing integer arrays. The structures were extensions to an address-calculation coding scheme [19] that is based on rather tight estimates of code lengths. In fact, the indexing technique that we use can be applied to other coding schemes as well. The main theoretical contribution of this paper was the improvement of the complexity of the access operation to $O(\lg \lg(n + s))$. The practical improvement was that, for the test data sets, the new access times were less than half of the old access times. Naturally, if we knew that our elements are uniformly distributed within a given interval, variable-length coding would not pay off. On the other hand, for fixed-length coding, a single outlier ruins the compactness of the code.

From the coding-efficiency point of view, as for the basic address-calculation method, our methods are compact (with overhead that is linear in n). Theoretically, the so-called succinct structures are more space-effective (with sublinear overhead), but usually harder to implement and less versatile with respect to the supported operations. Our aim was to keep the implementations practical, i.e. programmable with reasonable effort and with small constant factors in lower-order terms. Experimental results indicate that our indexed address-calculation coding for static arrays improves over the basic address-calculation coding, which has already demonstrated very good practical performance compared to the other compact coding methods [2, 4, 14, 20]. According to experiments, our structures are suitable for applications that need access to restricted subsets of elements at a time and possibly performing occasional updates.

Future work would include investigating both the theoretical complexities and practical usefulness of related methods; especially interesting is the trade-off between the overhead and access times. Practical experiments should be extended by using data from real applications such as inverted indexes within information retrieval systems.

References

1. Agrawal, R., Imieliński, T., Swami, A.: Mining association rules between sets of items in large databases. SIGMOD Rec. 22(2), 207–216 (1993)
2. Brisaboa, N.R., Ladra, S., Navarro, G.: Directly addressable variable-length codes. In: Proc. 16th Symp. on String Process. and Inform. Retrieval. Lecture Notes in Comput. Sci., vol. 5721, pp. 122–130. Springer-Verlag, Berlin/Heidelberg (2009)
3. Brodnik, A.: Computation of the least significant set bit. In: Proc. Electrotechnical and Comput. Sci. Conf. vol. B, pp. 7–10 (1993)
4. Culpepper, J.S., Moffat, A.: Compact set representation for information retrieval. In: Proc. 14th Symp. on String Process. and Inform. Retrieval. Lecture Notes in Comput. Sci., vol. 4726, pp. 137–148. Springer-Verlag, Berlin/Heidelberg (2007)
5. Delpratt, O., Rahman, N., Raman, R.: Compressed prefix sums. In: Proc. 33rd Conf. on Current Trends in Theory and Practice of Comput. Sci. Lecture Notes in Comput. Sci., vol. 4362, pp. 235–247. Springer-Verlag, Berlin/Heidelberg (2007)
6. Fenwick, P.M.: A new data structure for cumulative frequency tables. Software Pract. Exper. 24(3), 327–336 (1994)
7. Ferragina, P., Venturini, R.: A simple storage scheme for strings achieving entropy bounds. Theoret. Comput. Sci. 372(1), 115–121 (2007)
8. González, R., Navarro, G.: Statistical encoding of succinct data structures. In: Proc. 17th Annual Symp. on Combinatorial Pattern Matching. Lecture Notes in Comput. Sci., vol. 4009, pp. 294–305. Springer-Verlag, Berlin/Heidelberg (2006)
9. Gupta, A., Hon, W.K., Shah, R., Vitter, J.S.: Compressed data structures: Dictionaries and data-aware measures. Theoret. Comput. Sci. 387(3), 313–331 (2007)
10. Hagerup, T.: Sorting and searching on the word RAM. In: Proc. 15th Annual Symp. on Theoret. Aspects of Comput. Sci. Lecture Notes in Comput. Sci., vol. 1373, pp. 366–398. Springer-Verlag, Berlin/Heidelberg (1998)
11. Jansson, J., Sadakane, K., Sung, W.K.: CRAM: Compressed random access memory. E-print arXiv:1011.1708v2, arXiv.org, Ithaca (2012)
12. Katajainen, J., Rao, S.S.: A compact data structure for representing a dynamic multiset. Inform. Process. Lett. 110(23), 1061–1066 (2010)
13. Moffat, A.: Compressing integer sequences and sets. In: Encyclopedia of Algorithms, pp. 178–182. Springer Science+Business Media, LLC, New York (2008)
14. Moffat, A., Stuiver, L.: Binary interpolative coding for effective index compression. Inform. Retrieval 3(1), 25–47 (2000)
15. Moffat, A., Zobel, J.: Self-indexing inverted files for fast text retrieval. ACM Trans. Inform. Syst. 14(4), 349–379 (1996)
16. Raman, R., Raman, V., Rao, S.S.: Succinct dynamic data structures. In: Proc. 7th Int. Workshop on Algorithms and Data Structures. Lecture Notes in Comput. Sci., vol. 2125, pp. 426–437. Springer-Verlag, Berlin/Heidelberg (2001)
17. Raman, R., Raman, V., Satti, S.R.: Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. ACM Trans. Algorithms 3(4), 43:1–43:25 (2007)
18. Sadakane, K., Grossi, R.: Squeezing succinct data structures into entropy bounds. In: Proc. 17th Annual ACM-SIAM Symp. on Discrete Algorithms. pp. 1230–1239. ACM/SIAM, New York/Philadelphia (2006)
19. Teuhola, J.: Interpolative coding of integer sequences supporting log-time random access. Inform. Process. Manag. 47(5), 742–761 (2011)
20. Transier, F., Sanders, P.: Engineering basic algorithms of an in-memory text search engine. ACM Trans. Inform. Syst. 29(1), 2:1–2:37 (2010)