Multilevel Buckets for Sequential Decoding of Polar Codes

Nikolai Iakuba, Peter Trifonov
Distributed Computing and Networking Department
Peter the Great St. Petersburg Polytechnic University
Email: {nyakuba,petert}@dcn.icc.spbstu.ru

***Abstract*—The problem of efficient decoding of polar codes is considered. A multilevel bucket structure is proposed for implementation of the stack in the sequential decoding algorithm. This data structure reduces the complexity of stack operations with respect to the implementation based on a red-black tree.**

I. Introduction

Polar codes were recently shown to be able to achieve the capacity of a wide class of communication channels [1]. However, the performance of moderate length polar codes under the successive cancellation decoding algorithm appears to be quite poor. This problem was addressed in [2] where a list decoding algorithm was introduced. It was shown in [3], that the same performance can be achieved with much lower complexity by employing a stack-based decoding algorithm. Further complexity reduction can be obtained by employing the sequential decoding algorithm presented in [4]. It uses the information about the quality of non-yet-processed frozen bit subchannels to reduce the probability of selection of an incorrect path within the code tree. Additionally, it enables one to perform decoding in the log-likelihood ratios (LLR) domain using only comparison and summation operations.

The sequential decoding algorithm requires one to maintain a stack (priority queue), which can efficiently return the paths with maximal and minimal metrics. The implementation of the stack has significant impact on the complexity of the overall decoding algorithm. The straightforward approach to construct the stack, which can support the operations required by the sequential decoder, is to use a red-black tree data structure.

In this paper we describe a more efficient method, which is based on a multilevel bucket data structure (MBDS). The multilevel bucket data structure was invented by Denardo and Fox, and developed further by Ahuja et al. [5]. Its implementation was further simplified in [6]. MBDS was originally suggested for the case of monotone sequences of operations over elements with non-negative integer keys. It enables very efficient solution of the shortest path problem.

However, MBDS cannot be immediately used in the sequential decoding algorithm, since it generates a non-monotone sequence of operations. In this paper we propose some modifications to MBDS, which enable its application in the sequential decoder, and show that this results in substantially smaller number of comparison operations with respect to the case of the stack implementation based on a red-black tree.

The paper is organized as follows. Section II provides a review of polar codes, sequential decoding and MBDS. The proposed implementation of the stack for sequential decoding of polar codes is presented in Section III. Numeric results illustrating the complexity of the proposed approach are given in Section IV.

II. Background

A. Polar codes

An $(n = 2^m, k, d)$ polar code is a binary linear block code generated by some k rows of matrix $G_n = B_n A^{\otimes m}$, where n denotes the length of the code, k is the number of information symbols in a codeword, d is the minimal distance of the code, B_n is a bit-reversal permutation matrix, $A = \left(\begin{smallmatrix} 1 & 0 \\ 1 & 1 \end{smallmatrix}\right)$ and $\otimes m$ denotes m-fold Kronecker product.

By a_i^j we denote the row vector $(a_i, a_{i+1}, \ldots, a_j)$. Any codeword can be represented as $c_0^{n-1} = u_0^{n-1} G_n$, where vector $u_0^{n-1} \in \{0,1\}^n$ is an input sequence, such that u_ϕ contains information symbol if $\phi \notin \mathcal{F}$, and $u_\phi = 0$ (frozen symbol) otherwise. $\mathcal{F} \subset \{0, \ldots, n-1\}$ is the set of $n-k$ indices of frozen symbols.

It was suggested in [7] to set frozen symbols not to 0, but to some linear combinations of other symbols of u_0^{n-1}. Such codes, which are known as polar codes with dynamic frozen symbols, provide higher minimal distance d compared to classical polar codes. This results in better performance under list and sequential decoding.

B. The sequential decoding algorithm

In this section we present a new derivation of the sequential decoding algorithm, described in [4]. The objective of the decoding algorithm is to find a codeword u_0^{n-1} maximizing probability $P(u_0^{n-1}|y_0^{n-1})$, where y_0^{n-1} is the received noisy sequence. Let $a_{i,\mathcal{D}}^j$ denote the subvector of a_j^i consisting of elements $a_s, s \in \mathcal{D} \cap \{i, \ldots, j\}$. The sequential decoder estimates the probability

$$T(u_0^\phi, y_0^{n-1}) = \max_{u_{\phi+1}^{n-1} : u_{\phi+1,\mathcal{F}}^{n-1} = 0} P(u_0^{n-1}|y_0^{n-1}) \qquad (1)$$

of transmission of the most probable codeword of the polar code, given by prefix u_0^ϕ, assuming that the values of u_0^ϕ are correct. In log-domain this value can be approximated by

$$\hat{T}(u_0^\phi, y_0^{n-1}) = \ln R(u_0^\phi|y_0^{n-1}) + \hat{\Omega}_{ln}(\phi), \qquad (2)$$

where

$$R(u_0^\phi|y_0^{n-1}) = \max_{u_{\phi+1}^{n-1}} P(u_0^{n-1}|y_0^{n-1})$$

The $\hat{\Omega}_{ln}(\phi)$ value is an estimate of the probability of the event that the most probable path u_0^{n-1} with prefix u_0^{ϕ} satisfies the freezing constraints on symbols $u_j, j \in \mathcal{F}, j > \phi$. The values

$$\hat{\Omega}_{ln}(\phi) = \sum_{j>\phi} \log(1 - P_j),$$

where P_j is the bit error probability in the j-th subchannel $P(y_0^{n-1}, u_0^{j-1}|u_j)$ of the polarizing transformation, depend only on channel properties and can be precomputed offline. Thus, on each decoder iteration we need to compute only $\hat{R}(u_0^{\phi}, u_{\phi+1}|y_0^{n-1}) = \ln R(u_0^{\phi}, u_{\phi+1}|y_0^{n-1})$.

To obtain an efficient implementation of the sequential decoding algorithm we follow [2]. Let $\lambda \in [0, m]$, $\phi \in [0, n-1]$ and $\beta \in [0, 2^{m-\lambda} - 1]$ denote layer, phase and branch number respectively. Following [4], let us define the log-likelihood ratios (LLR)

$$S_m^{\phi}[0] = \ln\left(\frac{R(u_0^{\phi}, 0|y_0^{n-1})}{R(u_0^{\phi}, 1|y_0^{n-1})}\right),$$

which can be computed as

$$S_{\lambda}^{\phi}[\beta] = \begin{cases} Q(S_{\lambda-1}^{\phi}[2\beta], S_{\lambda-1}^{\phi}[2\beta+1]), & \phi \text{ even} \\ (-1)^{C_{\lambda}[\beta][0]} S_{\lambda-1}^{\phi}[2\beta] + S_{\lambda-1}^{\phi}[2\beta+1], & \phi \text{ odd,} \end{cases} \tag{3}$$

where $Q(a, b) = (-1)^{\mathrm{sgn}(a)+\mathrm{sgn}(b)} \cdot \min(|a|, |b|)$,

$$\mathrm{sgn}(a) = \begin{cases} 0, a > 0 \\ 1, a < 0 \end{cases},$$

and $C_{\lambda}[\beta][0]$ are the intermediate values of the polarizing transformation [2]. The initial values for this recursion are given by $S_0[j] = \ln(\frac{P(0|y_j)}{P(1|y_j)})$.

Consider $\hat{u}_{\phi+1}^{n-1} = \arg\max_{u_{\phi+1}^{n-1}} P(u_0^{n-1}|y_0^{n-1})$, the subvector $u_{\phi+1}^{n-1}$ of the most probable input vector of the polarizing transformation with prefix u_0^{ϕ}. Observe that $R(u_0^{\phi}, \tilde{u}_{\phi+1}|y_0^{n-1}) = R(u_0^{\phi}|y_0^{n-1})$ for $\tilde{u}_{\phi+1} = \hat{u}_{\phi+1}$. For $\tilde{u}_{\phi+1} = 1 - \hat{u}_{\phi+1}$ one obtains

$$\begin{aligned} \hat{R}(u_0^{\phi}, \tilde{u}_{\phi+1} = 1 - \hat{u}_{\phi+1}|y_0^{n-1}) &= \ln(P(u_0^{\phi}, \tilde{u}_{\phi+1}|y_0^{n-1}) \cdot \\ &\max_{u_{\phi+2}^{n-1}} P(u_{\phi+2}^{n-1}|u_0^{\phi}, \tilde{u}_{\phi+1}, y_0^{n-1}) = \\ &\hat{R}(u_0^{\phi}, \hat{u}_{\phi+1}|y_0^{n-1}) + \ln\left(\frac{R(u_0^{\phi}, \tilde{u}_{\phi+1}|y_0^{n-1})}{R(u_0^{\phi}, \hat{u}_{\phi+1}|y_0^{n-1})}\right). \end{aligned}$$

Hence one can compute $\hat{R}(u_0^{\phi}, u_{\phi+1}|y_0^{n-1})$ as

$$\hat{R}(u_0^{\phi}, u_{\phi+1}|y_0^{n-1}) = \begin{cases} \hat{R}(u_0^{\phi}|y_0^{n-1}), \text{if } u_{\phi+1} = \mathrm{sgn}(S_m^{\phi}[0]) \\ \hat{R}(u_0^{\phi}|y_0^{n-1}) - |S_m^{\phi}[0]|, \text{ otherwise.} \end{cases} \tag{4}$$

One can assume $\hat{R}(\epsilon|y_0^{n-1}) = 0$, where ϵ is a zero-length vector.

The decoder maintains a stack (priority queue) with capacity Θ, that contains paths $u_0^{\phi-1}$ in the code tree and corresponding metrics $\hat{T}(u_0^{\phi-1}, y_0^{n-1})$. At each iteration the decoder extracts from the stack a path with the highest metric. If $\phi \in \mathcal{F}$, it extends the path with $u_{\phi} = 0$ (for classical polar codes), computes the metric for the extended path, and pushes it into the stack.

Otherwise, two copies of the path are constructed, which are extended with zero and one. The decoder calculates the corresponding metrics, and pushes two extended paths into the stack. If the number of paths in the stack is about to exceed Θ, the paths with minimal metric values are removed from it.

Also the decoder maintains a vector $q = (q_0, \ldots, q_{n-1})$, where q_{ϕ} is the number of times the decoder has visited phase ϕ, i.e. some path $u_0^{\phi-1}$ of length ϕ was extracted from the stack. If q_{ϕ} exceeds some threshold L, all paths $u_0^{\psi-1}, \psi \leq \phi$, are removed from the stack. Here L and Θ are the parameters of the decoder, which affect its complexity and memory requirements, as well as the decoding error probability.

Hence, the stack used in the sequential decoding algorithm must support the following operations:

- $Push(T, u_0^{\phi})$: insert path u_0^{ϕ} with metric T into the stack.
- $PopMax()$: return a path with the maximal metric and remove it from the stack.
- $PopMin()$: return a path with the minimal metric and remove it from the stack.
- $Erase(\phi)$: remove all paths $u_0^{\psi-1}$ with $\psi \leq \phi$ from the stack.

C. Multilevel bucket data structure

The multilevel bucket data structure (MBDS) is a monotone data structure, that supports $Insert$ and $ExtractMin$ operations. These operations insert into the MBDS an element with a non-negative integer key, and extract an element with the smallest key, respectively. The keys are assumed to be in range $[0, C]$.

A sequence of operations of a data structure is called monotone if the corresponding sequence of element keys returned by $ExtractMin$ is non-decreasing. The data structures that perform such sequences of operations correctly are called monotone.

Here we give a simplified description of the MBDS [6]. The structure consists of t levels and $\Delta = \lceil C^{1/t} \rceil$ buckets on each level. Every bucket is represented by a single-linked list, containing elements and their keys. We number levels from 1 to t, and refer to the level 1 as the lowest level in the structure. The operations performed on the MBDS cause the elements to be distributed between buckets, such that the keys of elements on lower levels are less than the keys of elements on higher levels.

Let $v(u)$ and μ be a key of some element u in the MBDS, and the key of the last element returned by $ExtractMin$, respectively. For any element u with key $v = v(u)$ one can determine its position relatively to μ as follows. Let $v = \sum_{s=1}^{t} v_s \Delta^{s-1}$ be the base-Δ representation of v, where v_s is the s-th digit. An element u is stored in bucket $B(i, j)$ iff $i = \max_{v_s \neq \mu_s} s$ and $j = v_i$ where $B(i, j)$ denotes the j-th bucket on the i-th level of MBDS. In other words, i is a position of

INSERT(u)
1 $(i,j) \leftarrow$ CALCULATEPOSITION($\mu, v(u)$)
2 insert u into $B(i,j)$

Fig. 1. Insertion of element into the MBDS

EXTRACTMIN()
1 find the lowest non-empty level i
2 find first non-empty bucket j on level i
3 find the element u with minimal key in $B(i,j)$
4 extract u from $B(i,j)$
5 $\mu \leftarrow v(u)$
6 **if** $i > 1$
7 **then for** each element u in $B(i,j)$
8 **do** $(i',j') \leftarrow$ CALCULATEPOSITION($\mu, v(u)$)
9 insert u into bucket $B(i',j')$
10 **return** u

Fig. 2. Extraction of an element with minimal key from the MBDS

the most significant digit in which v and μ differ, and j is the value of this digit of the key.

Example 1. Let $\Delta = 10, \mu = 1145\underline{3}26$ and $u = 1145\underline{9}00$. It can be seen that the most significant digit where u and μ differ is the third one. Thus the position of u relatively to μ is $(i = 3, j = 9)$.

By choosing the parameter

$$\Delta = 2^{\lceil (\log_2 C)/t \rceil} \tag{5}$$

and using the random access memory (RAM) model of computations one can calculate the element position in constant time. Let $CalculatePosition(\mu, z)$ be the function which finds such values i, j. Figures 1 and 2 present a description of $Insert$ and $ExtractMin$ operations of the multilevel bucket data structure.

In order to search efficiently for the first non-empty level in the structure, one can maintain an array of flags, that indicates if the corresponding level is not empty. Using binary search, one can find the indices of the highest and the lowest non-empty levels in $O(\log k)$ time. Alternatively, it is possible to find the index of the lowest non-empty level in constant time using the power of the RAM model.

III. STACK IMPLEMENTATION BASED ON THE MULTILEVEL BUCKET DATA STRUCTURE

We propose to implement the stack needed by the sequential decoder using the MBDS. However, some modifications are required, since the decoder needs to be able to find the paths both with minimal and maximal metrics, and delete the elements from the stack. Furthermore, MBDS operates with non-negative integer keys, while path metrics $\hat{T}(u_0^i, y_0^{n-1})$ can take arbitrary real values.

A. From path metrics to integer keys

The real-valued path metrics given by (2) should be mapped onto non-negative integers, which can be used as keys in the MBDS. We propose to define this mapping as

$$v(u_0^\phi) = \max\left(0, M - \lceil \hat{T}(u_0^\phi, y_0^{n-1}) \cdot a \rceil\right), \tag{6}$$

where M is a sufficiently large integer.

This mapping can be implemented by multiplying input LLR values by a, rounding the result of this operation to an integer, and performing calculations given by (3)–(4) using integer arithmetic. Then one can obtain path key as

$$v(u_0^\phi) = \max\left(0, \Omega'[\phi] - \hat{R}(u_0^\phi | y_0^{n-1})\right), \tag{7}$$

where $\Omega'[\phi] = M - \lceil \hat{\Omega}_{ln}[\phi] \cdot a \rceil$.

Applying such mapping may cause two paths with different metrics to be assigned the same key value. If one of these paths is a correct one, a decoding error may occur. Hence, a must be a sufficiently large number. In practice, it should be selected so that the quantized values $[S_0[j] \cdot a]$ fit into a b-bit integer.

Expression (6) ensures that path keys are non-negative. The value of M should be selected so that the probability of obtaining a path with zero key $v(u_0^\phi)$ is sufficiently small. Consider the case of AWGN channel with noise power $N_0 = 2\sigma^2$. Observe that $\hat{T}(u_0^\phi, y_0^{n-1})$ is a linear combination of at most n LLR values $S_0[j]$ with coefficients in $\{-1, 1\}$. Furthermore, it is an estimate of the final path metric $\hat{T}(u_0^{n-1}, y_0^{n-1})$. For a correct decoding path, $\hat{T}(u_0^{n-1}, y_0^{n-1}) \cdot a$ is a Gaussian random variable with mean value $2an/\sigma^2$ and variance $4a^2n/\sigma^2$. Hence, $P\{\hat{T}(u_0^{n-1}, y_0^{n-1}) \cdot a > M\} = \frac{1}{2} - \Phi_0(\frac{M - 2an/\sigma^2}{2a\sqrt{n}/\sigma})$, where

$$\Phi_0(z) = \frac{1}{\sqrt{2\pi}} \int_0^z e^{-t^2/2} \mathrm{d}t$$

One should set M so that a zero key appears with sufficiently small probability p, i.e.

$$M = 2na/\sigma^2 + 2\sqrt{n}a\Phi_0^{-1}(1/2 - p)/\sigma. \tag{8}$$

Hence one can set the value $C = 2M$ and, taking into account (5), find values t and Δ which minimize the average decoding complexity. Unfortunately, we do not have an analytic expression for the average complexity of MBDS operations in the case of sequential decoding. Hence, this optimization requires one to do simulations.

B. Dealing with non-monotone sequence of operations

Due to transformation (6), the decoder needs to extract at each iteration a path with the smallest key $v(u_0^\phi)$. However, the sequence of such keys in the considered case of sequential decoding of polar codes is non-monotone. That is, after extraction of an element with key μ, the decoder may push into the stack elements with smaller keys.

In order to support non-monotone sequences of operations, we introduce an auxiliary level 0 into the MBDS. This level contains a single bucket $B(0,0)$. Paths with keys $v(u_0^\phi) < \mu$, where μ is the smallest path key extracted up to now, are stored in $B(0,0)$, as shown in Figure 3. This bucket can be implemented in the same way as other buckets, or using any standard priority queue data structure, like Fibonacci heap.

If $B(0,0)$ is empty, the proposed stack implementation operates as a classical MBDS. Otherwise, the path u with the

PUSH(u)
1 **if** $v(u) < \mu$
2 **then** insert u into $B(0,0)$
3 **else** INSERT(u)

Fig. 3. Insertion of a path into MBDS-based stack

POPMAX()
1 **if** $B(0,0)$ is empty
2 **then** $u \leftarrow$ EXTRACTMIN()
3 **else** find minimal $v(u)$ in $B(0,0)$
4 extract u from $B(0,0)$
5 **if** $l_0 \geq l$
6 **then** $\mu' \leftarrow \mu$
7 $(i,j) \leftarrow$ CALCULATEPOSITION(μ, μ')
8 $\mu \leftarrow v(u)$
9 UPDATELOWLEVELS(i,j)
10 **return** u

Fig. 4. Extracting a path with the highest metric from the MBDS-based stack

smallest key $v(u)$ is extracted from $B(0,0)$. If, after extraction of the path from $B(0,0)$, the number of elements l_0 in it is still at least l, then the value of μ is changed, and the elements stored in the stack are rearranged in order to obtain a proper MBDS.

Let i be the position of the most significant digit where μ and $v(u)$ differ, and j be the value of this digit in the base-Δ representation of μ. Observe, that the keys of elements, stored on levels $1, 2, \cdots, i-1$, have the same value of the i-th digit in base-Δ representation as μ. Therefore all elements stored on the specified levels should be moved to the $B(i,j)$. This operation can be performed by merging of the corresponding linked lists. Finally, the remaining elements stored in $B(0,0)$ should be inserted into an appropriate bucket $B(i,j), i > 0$, and removed from $B(0,0)$.

Figure 4 illustrates the proposed algorithm for finding the path with the highest metric, i.e. the smallest key. The proposed rearrangement algorithm is shown in Figure 5.

One needs to optimize the value of l in order to minimize the average number of operations performed by the proposed stack implementation. If l is set too high, then $B(0,0)$ may contain many elements, so that finding an element with the minimal key value becomes costly. If l is small, the relatively expensive $UpdateLowLevels$ operation is executed too often.

The $PopMin$ operation is similar to the $PopMax$ operation, except that we have to examine the highest non-empty level in the structure, as shown in Figure 6. This operation

UPDATELOWLEVELS(i,j)
1 **for** $i' = 1$ **to** $i-1$
2 **do for** each bucket j' on level i'
3 **do** move all elements from $B(i',j')$ to $B(i,j)$
4 **for** each element u in $B(0,0)$
5 **do** INSERT(u)

Fig. 5. Rearrangement of the data stored in the stack into a proper MBDS

POPMIN()
1 find maximal non-empty level index i
2 find maximal non-empty bucket index j on level i
3 Extract element u with maximal $v(u)$ from $B(i,j)$
4 **return** u

Fig. 6. Extracting a path with the lowest metric from the MBDS-based stack

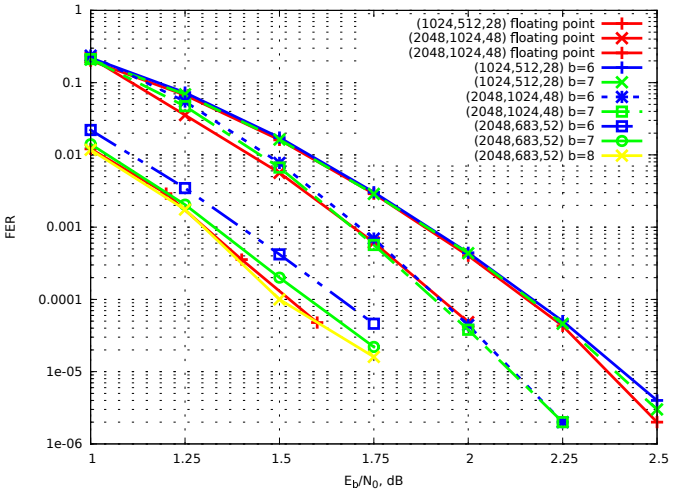

Fig. 7. Performance of the MBDS-based sequential decoder

does not require updating any element positions and the value of μ. Recall, that this operation is used by the sequential decoder in the case of stack overflow for removing paths with low scores. Bucket $B(i,j)$ selected at line 3 of the algorithm contains typically many elements with relatively close key (i.e. metric) values. In order to reduce the complexity, one can avoid searching for the element with maximal $v(u)$, and take an arbitrary one. Simulations show that this has negligible impact on the decoding error probability.

C. Removing short paths

In certain cases the sequential decoding algorithm needs to kill the paths shorter than some value ϕ. This requires removing the corresponding elements from the stack. In order to avoid searching for these elements in the MBDS, we propose to delay path removal. That is, we define an integer variable ϕ_0. As soon as one obtains $q_\phi > L$ at some phase ϕ, we propose to set $\phi_0 = \phi$. If a path $u_0^{\psi-1}, \psi \leq \phi_0$, is returned by $PopMax$ operation, then this path is just killed, and the decoder proceeds to the next iteration.

Delayed path removal may cause the stack to be filled with invalid paths. If the number of elements stored in the stack reaches Θ, then one should inspect all these elements, and kill those corresponding to the paths shorter than ϕ_0. This requires just removing them from the corresponding buckets. If no such paths are found, $PopMin$ operation should be used to remove the paths with the lowest metric.

IV. NUMERIC RESULTS

The proposed implementation of the stack requires only comparison and bit manipulation operations. Therefore, in this

TABLE I. NORMALIZED AVERAGE NUMBER OF COMPARISONS.

$(1024, 512, 28)$ code, $L = 32$, $\Theta = 5000$, $E_b/N_0 = 1$					
	$l = 1$	$l = 2$	$l = 3$	$l = 4$	$l = 5$
$t = 5, \Delta = 8$	1.0134	1.0010	1 (158337)	1.0005	1.0017
$t = 4, \Delta = 16$	1.0230	1.0024	1 (132524)	1.0002	1.0013
$t = 3, \Delta = 32$	1.0261	1.0033	1.0002	1 (153874)	1.0007
$(1024, 512, 28)$ code, $L = 32$, $\Theta = 5000$, $E_b/N_0 = 1.5$					
	$l = 1$	$l = 2$	$l = 3$	$l = 4$	$l = 5$
$t = 5, \Delta = 8$	1 (57959)	1.0001	1.0034	1.0076	1.0121
$t = 4, \Delta = 16$	1.0017	1 (52586)	1.0031	1.0076	1.0124
$t = 3, \Delta = 32$	1.0031	1 (56914)	1.0026	1.0065	1.0109
$(1024, 512, 28)$ code, $L = 32$, $\Theta = 5000$, $E_b/N_0 = 2$					
	$l = 1$	$l = 2$	$l = 3$	$l = 4$	$l = 5$
$t = 5, \Delta = 8$	1 (27039)	1.0059	1.0123	1.0188	1.0255
$t = 4, \Delta = 16$	1 (25868)	1.0060	1.0128	1.0197	1.0268
$t = 3, \Delta = 32$	1 (26818)	1.0056	1.0121	1.0187	1.0255
$(2048, 683, 52)$ code, $L = 32$, $\Theta = 5000$, $E_b/N_0 = 1.5$					
	$l = 1$	$l = 2$	$l = 3$	$l = 4$	$l = 5$
$t = 5, \Delta = 8$	1 (58612)	1.0031	1.0075	1.0134	1.0242
$t = 4, \Delta = 16$	1 (53105)	1.0038	1.0090	1.0157	1.0278
$t = 3, \Delta = 32$	1 (58098)	1.0030	1.0077	1.0139	1.0249
$(2048, 683, 52)$ code, $L = 32$, $\Theta = 5000$, $E_b/N_0 = 2$					
	$l = 1$	$l = 2$	$l = 3$	$l = 4$	$l = 5$
$t = 5, \Delta = 8$	1 (40201)	1.0059	1.0166	1.0324	1.0457
$t = 4, \Delta = 16$	1 (38765)	1.0065	1.0177	1.0341	1.0480
$t = 3, \Delta = 32$	1 (40034)	1.0063	1.0171	1.0329	1.0464

section we report the average number of comparison operations required by the sequential decoder for different parameters of the proposed data structure. We consider the case of transmission of BPSK-modulated codewords of $(1024, 512, 28)$, $(2048, 1024, 48)$ and $(2048, 683, 52)$ polar codes with dynamic frozen symbols [7] over the AWGN channel.

Figure 7 presents the performance of the sequential decoding algorithm based on the proposed stack implementation for different values of the number of quantization bits b. For comparison, we report also the results obtained with an implementation based on red-black tree and floating point arithmetic. It can be seen that for $b = 7$ the performance is almost the same as in the case of floating point arithmetics. It was shown in [8] that in the case of the successive cancellation decoding algorithm 5 or 6 quantization bits are sufficient to reach the decoding performance close to that of the floating point implementation of the decoder. Extra quantization bits are needed by the sequential decoder since, for a fixed decoding error probability, it operates at much lower SNR than the successive cancellation decoder, and is therefore more sensitive to quantization noise.

Let $\Psi(t, \Delta, l)$ be the average number of comparisons performed during the decoding process. Table I presents the normalized values $\frac{\Psi(t,\Delta,l)}{\min_l \Psi(t,\Delta,l)}$ for $(1024, 512, 28)$ and $(2048, 683, 52)$ codes, and different values $E_b/N_0, t, \Delta$. The values of $\min_l \Psi(t, \Delta, l)$ are shown in brackets.

It can be seen, that if the channel has good quality, then the optimal l value is 1 and the number of comparisons slightly increases with l. In the case of highly noisy channels the optimal l value increases, and choosing $l = 1$ leads to small overhead. Furthermore, the results, obtained for different values of t and Δ behave similarly. Hence, one can set $l = 1$, and optimize only t and Δ.

Figure 8 shows the average number of comparisons performed by the sequential decoder for the $(1024, 512, 28)$ code for the cases of the stack implemented using MBDS and a red-black tree, provided by the standard *C++* library. It can

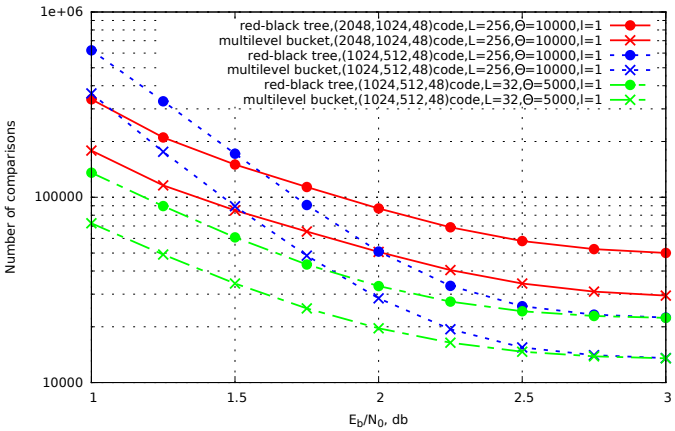

Fig. 8. An average number of comparisons.

be seen, that the average number of comparisons required by the MBDS-based decoder at $E_b/N_0 = 2$ dB is reduced by 40% compared to the case of the red-black tree.

V. CONCLUSION

In this paper an efficient implementation of the stack (priority queue) required by the sequential decoding algorithm for polar codes was presented. The proposed approach employs only bit manipulation and comparison operations to manage the paths considered by the decoder, and can be naturally used with the fixed-point arithmetic, which is required for hardware implementation. The average decoding complexity was shown to be less compared to the implementation based on a red-black tree.

REFERENCES

[1] E. Arikan, "Channel polarization: A method for constructing capacity-achieving codes for symmetric binary-input memoryless channels," *IEEE Transactions on Information Theory*, vol. 55, no. 7, pp. 3051–3073, July 2009.

[2] I. Tal and A. Vardy, "List decoding of polar codes," in *Proceedings of IEEE International Symposium on Information Theory*, 2011, pp. 1–5.

[3] K. Niu and K. Chen, "CRC-aided decoding of polar codes," *IEEE Communications Letters*, vol. 16, no. 10, pp. 1668–1671, October 2012.

[4] V. Miloslavskaya and P. Trifonov, "Sequential decoding of polar codes," *IEEE Communications Letters*, vol. 18, no. 7, pp. 1127–1130, 2014.

[5] R. K. Ahuja, K. Mehlhorn, J. B. Orlin, and R. E. Tarjan, "Faster algorithms for the shortest path problem," *Journal of ACM*, vol. 37, no. 2, pp. 213–223, 1990.

[6] B. Cherkassky, A. V. Goldberg, and C. Silverstein, "Buckets, heaps, lists, and monotone priority queues," in *Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms*, 1997.

[7] P. Trifonov and V. Miloslavskaya, "Polar codes with dynamic frozen symbols and their decoding by directed search," in *Proceedings of IEEE Information Theory Workshop*, September 2013, pp. 1 – 5.

[8] C. Leroux, A. J. Raymond, G. Sarkis, I. Tal, A. Vardy, and W. J. Gross, "Hardware implementation of successive-cancellation decoders for polar codes," *Journal of Signal Processing Systems*, vol. 69, no. 3, pp. 305–315, 2012.