White Paper

5G Technology
Radio Access Networks

Polar Decoder White Paper

Table of Contents

1 Executive Summary

A 5G base station must have the capability to decode polar codes to receive the data in the control channels. Polar codes introduce a new approach to the construction and decoding of the error correcting mechanisms that are widely used in wireless systems. This new approach requires a unique decoding procedure that is not well suited to highly parallel hardware architectures. Each control message must be decoded with a short latency. Highly parallel architectures sacrifice latency for decoding many messages concurrently to achieve high throughputs.

The codes used in the 3G and 4G control channels do not use polar codes and are decoded as part of the software stack in Intel® Xeon® processors, without using hardware accelerators. There is a strong motivation for the 5G control-channels to follow the same all-software scheme. A high performance polar list decoder that meets both the low latency and error-correction requirements has been achieved using Intel Xeon processors and the Intel® Advanced Vector Extensions 512 (Intel® AVX-512) instruction set.

Intel's FlexRAN 5GNR reference design software has been optimized for the Intel Xeon processor and contains a library of polar decoders that fulfill the 3GPP requirements for list decoding of polar codes of all lengths. Using this reference design, operators can take advantage of these libraries to decode polar codes and reduce their overall development schedule while accelerating time to deployment for 5G virtualized base stations.

1.1 Background

Polar codes, discovered in 2008 by Erdal Arikan, have been selected by 3GPP for use in the 5G Wireless Standard. Polar codes form a class of error correcting channel codes that are used to transform the message data into a codeword that is longer than the message itself. Using the extra data in the codeword, special hardware or software can recover the message that was transmitted from the received data—data that has been corrupted by noise and interference.

Techniques for decoding polar codes have been developed that demonstrate their superiority over the methods used in the 3G and 4G Wireless Standards. Using advanced techniques such as list decoding, the polar codes used for the messages in the control channels are able to correct errors in noisier environments, thus improving the sensitivity of the wireless receivers and increasing the data throughput.

The list decoding required for polar codes is a form of recursive tree search with a low level of exploitable parallelism that is defined by the list size. The list sizes are small—in the region of 8,16 or 32—and so the usefulness of massive parallelism is limited. However, the Intel Xeon processor with Intel AVX-512 instruction set, which offers vector data types of lengths 64,32,16, and 8, is an excellent match for list decoding and the 5G control channels can still be decoded as part of the wireless software stack with no additional hardware acceleration or specialized processor architectures.

Recursive tree search allows a very compact description of the decoder software, but this can hide some important issues. In particular, the decoder is best thought of as a bit-by-bit estimation process, and each bit depends on the previous bits. A massively parallel machine is not a good match for a polar code as these data dependencies lead to sequential decoding steps.

2 Document Summary

This document describes the structure and performance of a fast recursive polar list decoder implemented using the Intel AVX-512 instruction set. This work was motivated by the requirement to conduct control channel decoding for 5GNR in software, maintaining equivalence to the 3G and 4G wireless control channels.

Polar codes are implicitly recursive in their conception and structure. In turn, a recursive decoder structure is a compact way to describe and write code for a polar decoder. In the example described in this document, the recursion was achieved using templated meta-programming with the Intel® C++ Compiler.

Due to the recursive nature of successive cancellation polar decoding, there is a strong data dependency between each recursion. This data dependency makes Intel Xeon processors an ideal fit for performing the decoding, since they possess advanced features that permit the efficient execution of the code. These features include out-of-order execution, branch prediction, register renaming, and an efficient cache structure. In addition, the Intel AVX-512 instruction naturally enables SIMD (Single Instruction Multiple Data) processing of a polar list decoder.

New instructions introduced in the third generation Intel Xeon Scalable processors accelerate the list decoding even further.

By exploiting the features of the Intel Xeon processors, fast 3GPP-compliant Simplified Successive Cancellation List decoders have been written, ensuring that control messages in the 5GNR standard may be decoded in software as they can for 3G and 4G. A parity-check (PC) version has also been proven for very short messages in the 3GPP 5GNR standard.

2.1 Terminology

Table 2 1: Terminology.

Term	Description
CRC	Cyclic Redundancy Check
LLR	Log Likelihood Ratio
SC	Successive Cancellation
SIMD	Single Instruction Multiple Data
SSC	Simplified Successive Cancellation

2.2 Reference Documents

Table 2 1: Reference Documents

Reference	Document	Document No./Location
[1]	A Brief Introduction to Polar Codes Notes for Introduction to Error-Correcting Codes Henry D. Pfister October 8th, 2017	polar_pdf
[2]	3GPP TS 38.212v15.0.0	TS38_212_v15
[3]	Intel Intrinsics Guide	Intel Intrinsics Guide
[4]	Fast List Decoders for Polar Codes: Sarkis, Vardy, Thibeault, and Gross	IEEE Journal On Selected Areas in Communications, vol. 34, no. 2, Feb 2016

3 Simplified Successive Cancellation List Decoding using Recursion

3.1 Introduction and Code Outline

Polar List Decoding can be encoded in a compact way using recursions. The recursive approach is efficient when implemented on Intel Xeon processors, as they have features that are beneficial for handling data dependencies, such as out-of-order execution, register renaming, and branch prediction.

Even in Intel processors, it is advantageous to avoid branches and conditional code as much as possible. When a list decoder is initialized, it starts with an empty list of candidate partial codewords. After the first binary decision, the list contains two entries (the "true" and "false" decisions), and four entries after the second decision, etc. The list would grow exponentially without pruning to some maximum list depth. In the decoder example discussed here, the list is pruned to 8 entries.

Using Intel's AVX-512 instruction set, the list LLRs and previous binary decisions can be stored in fixed-length vectors. It is advantageous, then, to arrange the decoder so that it operates on a list depth of 8 at all times with some special handling of the first three decisions. After these first three decisions, the decoder then runs unconditionally using Intel AVX-512 vectors to represent the list LLRs and list decisions.

An outline of the recursive list decoder is contained in Table 3 1. This outline is written in MATLAB®, which is the tool that was used for the test benching and development of the C++ code.

Table 3-1: Polar-SC-List-8 Recursion in MATLAB

```
global maxListDepth;
maxListDepth = 8;
global accumMetric;
global decisionStep;

function [s, list] = recurseNodeNatural8(llr, f)
% llr = input LLRs (8-by-CODEWORD_SIZE. If the input LLRs are (1-by-CODEWORD_SIZE) then use repmat(llrs, 8, 1)
% f = array indicating frozen bits. f(n)==1 to indicate that bit#n is frozen.
% s = output msg
% l = output LLRs.
%Stop recursions at node#0 (length-1)
N = size(llr, 2);
if (N==1)
    if (f == 0)
        %-ve LLR -> 1, +ve LLR -> 0
        %Decisions with no metric growth
        st = (llr <= 0);
        %metric for correct decision
        mt = zeros(size(llr));
        %Decisions with some metric growth
        sf = (llr > 0);
        %metric for forced error
        mf = abs(llr);
        %These if-elses need to be extended for list 16, 32, etc.
        %This example is for list 8.
        %This builds up all possible decisions in the first 3 unfrozen
        %branches, to fill up the list depth of 8.
        if (decisionStep == 0)
            s = [st(1:4); sf(5:8);];
            accumMetric = accumMetric + [mt(1:4); mf(5:8);];
            decisionStep = 1;
            list = 1:maxListDepth;
        elseif (decisionStep == 1)
            s = [st(1:2); sf(3:4); st(5:6); sf(7:8);];
            accumMetric = accumMetric + [mt(1:2); mf(3:4); mt(5:6); mf(7:8);];
            decisionStep = 2;
            list = 1:maxListDepth;
        elseif (decisionStep == 2)
            s = [st(1); sf(2); st(3); sf(4); st(5); sf(6); st(7); sf(8);];
            accumMetric = accumMetric + [mt(1); mf(2); mt(3); mf(4); mt(5); mf(6); mt(7); mf(8);];
            decisionStep = 3;
            list = 1:maxListDepth;
```

```
        else
            tmp = [accumMetric+mt; accumMetric+mf;];
            %Call the pruning function
            idx = medianMetricPrune(tmp);
            %assemble the possible symbols
            tmps = [st; sf;];
            s = tmps(idx);
            %Accumulate the metric
            accumMetric = tmp(idx);
            %Maintain a metric of zero at the top of the list
            accumMetric = accumMetric - min(accumMetric);
            %Update the list of which paths were followed.
            list = idx;
            %From the structure of tmp, entry (Z+maxListDepth) is path#Z,
            %etc.
            list(list > maxListDepth) = list(list > maxListDepth) - maxListDepth;
        end
    else
        %Frozen bit
        s = zeros(size(llr));
        %Update the metrics. The frozen bit is known to be 0 (-ve LLR)
        %so increase the metric where the LLR is +ve
        accumMetric = accumMetric + abs(llr) .* (llrs <= 0);
        %No list pruning necessary
        list = 1:maxListDepth;
    end
else
    %Compute prior-LLRs backwards (no knowledge of bits)
    idx1 = 1:N/2;
    idx2 = idx1 + N/2;
    llr1 = llr_prior(llr(:, idx1), llr(:, idx2));
    %Upper (left) branch
    [s1, list1] = recurseNodeNatural8(llr1, f(1:(N/2)), metricPruneHandle, metricHandle);
    %Got the decisions, so update LLRs based on these
    llr2 = llr_posterior(s1, llr(list1, idx1), llr(list1, idx2) );
    %lower (right) branch
    [s2, list2] = recurseNodeNatural8(llr2, f((N/2+1):end), metricPruneHandle, metricHandle);
    %These decisions go back up one node.
    list = list1(list2);
    %Modulo-2 (EXOR) addition
    s = [mod(s1(list2,:) + s2, 2) s2];
end

function z = llr_prior(w1,w2)
global maxListDepth;
for n=1:maxListDepth
    z(n,:) = sign(w1(n,:).*w2(n,:)) .* min(abs([w1(n,:); w2(n,:);]), [], 1);
end

function z = llr_posterior(s, w1, w2)
global maxListDepth;
for n=1:maxListDepth
    z(n,:) = (1-2*s(n,:)).*w1(n,:) + w2(n,:);
end

%Metric prune function: median threshold is equivalent to "best-N"
function idx = medianMetricPrune(metricsIn)
global maxListDepth;
%Sort for best metrics
[~, idx] = sort(metricsIn, 'ascend');
medianThresh = 0.5*(metricsIn(idx(maxListDepth)) + metricsIn(idx(maxListDepth+1)));
idx = find(metricsIn <= medianThresh, maxListDepth, 'first');
```

3.2 Preparation of Inputs for Intel AVX-512 Vectorization

The code is vectorized, meaning that the LLR vector llr is of dimension $8 \times N_{LLR}$ for list-8 decoders. Using the Intel AVX-512 instruction set, the SIMD-8 LLR vectors will be efficiently handled using native data types.

In both the MATLAB and the C++ code, the input LLRs are repeated 8 times. This means that, in a length-4 example, the input LLRs are repeated in this way:

$$\begin{matrix} llr_0 & llr_1 & llr_2 & llr_3 \\ llr_0 & llr_1 & llr_2 & llr_3 \\ llr_0 & llr_1 & llr_2 & llr_3 \\ llr_0 & llr_1 & llr_2 & llr_3 \\ llr_0 & llr_1 & llr_2 & llr_3 \\ llr_0 & llr_1 & llr_2 & llr_3 \\ llr_0 & llr_1 & llr_2 & llr_3 \\ llr_0 & llr_1 & llr_2 & llr_3 \end{matrix}$$

Each row in this table corresponds to a different list being decoded and each column corresponds to the set of LLRS for one message bit position. This is the initial input condition for SIMD-8/list-8 operation, and the purpose is to prepare the decoder inputs so that it can begin execution with list-8 style operation, allowing the C++ code to be specialized for all-list-8 processing.

3.3 C++ Code Using Recursive Template Meta-Programming

The C++ code is based on recursive meta templates. This allows the compiler to flatten code and unroll loops using the compile-time information provided. The basic C++ structure is replicated below in Table 3-2.

Table 3-2: Polar-SC-List-8 Recursion in C++

```
// Recursive Polar List Decoder Outline Code

// Recursions, with AVX512 specialization
template<unsigned NUM_LLRS>
struct PolarListRecursiveInt16
{
  // Bifurcation of the decoding tree
  static constexpr unsigned halfLlrs = NUM_LLRS / 2;
  // List handling class is returned
  static ListStructure
  Recurse(const __m128i* __restrict llrs_in, const SimdBitset<NUM_LLRS>& frozenSequence,
          BitList* __restrict message, BitList* __restrict codeword, Is16vec8& ref_metric,
          unsigned& ref_decision)
  {
    __m128i llrBuffer[halfLlrs];

    LlrPriorMinProdInt16<halfLlrs * 8>(llrs_in, llrs_in + halfLlrs, llrBuffer);

    ListStructure list1(IndexList::Default(), 0);

    if (frozenSequence.Lower().frozen())
    {
      // Call the frozen bit handling function
      list1 = FrozenNodeInt16(*llrBuffer, ref_metric);
    }
    else if (frozenSequence.Lower().unfrozen())
    {
```

```
      // Call the un-frozen bit handling function
      list1 = UnFrozenNodeInt16(*llrBuffer, message, codeword, ref_metric, ref_decision);
      // Prune the metrics
      PruneMetricsInt32(lowerMetric, upperMetric);
    }
    else
    {
      // Recurse again down one level
      list1 = PolarListRecursiveInt16<halfLlrs, ISA>::Recurse(llrBuffer, frozenUpper, message,
        codeword, ref_metric, ref_decision);
    }

    LlrPosteriorInt16<halfLlrs * 8>(codeword, llrs_in, llrs_in + halfLlrs, list1.list, llrBuffer);


    if (frozenSequence.Upper().frozen())
    {
      // Call the frozen bit handling function
      list2 = FrozenNodeInt16(*llrBuffer, ref_metric);
    }
    else if (frozenSequence.Upper().unfrozen())
    {
      // Call the un-frozen bit handling function
      list2 = UnFrozenNodeInt16(*llrBuffer, message + list1.msg_len, codeword + halfLlrs, ref_metric,
        ref_decision);
    }
    else
    {
      // Recurse again down one level
      list2 = PolarListRecursiveInt16<halfLlrs, ISA>::Recurse(llrBuffer, frozenLower,
        message + list1.msg_len, codeword + halfLlrs, ref_metric, ref_decision);
    }
    // Done the upper & lower for this branch: Polar Transform and merge lists
    // XOR for Polar Transform of List Entries
    XorInPlaceInt16<halfLlrs * 8>(codeword, codeword + halfLlrs, list2.list);
    // Merging of Lists
    MergeInPlaceInt16(message, list2.list, list1.msg_len);
    // Total message length in this pass comes from Upper+Lower
    return ListStructure(IndexList::ReorderIndexes(list1.list, list2.list),
                         list1.msg_len + list2.msg_len);
  }
};

// Instantiations
void
Int16Decoder(const Polar::DecoderRequest *request, Polar::DecoderResponse *response)
{
  const unsigned numLlrs = (unsigned) (1 << request->order);

  // Set-Up input LLRs and Frozen Bits

  // Instantiate each Recursive Template
  switch (numLlrs)
  {
  // Could include other sizes
  case 128:
    listOut = PolarListRecursiveInt16<128 >::Recurse(simdLlrs, SimdBitset<128>(*(const __m128i*)rawFrozenBits),
      messageLists, codewordLists, accumulatedMetric, decisionStage);
    break;
```

```
  case 256:
    listOut = PolarListRecursiveInt16<256 >::Recurse(simdLlrs, SimdBitset<256>(*(const __m256i*)rawFrozenBits),
      messageLists, codewordLists, accumulatedMetric, decisionStage);
    break;
  case 512:
    listOut = PolarListRecursiveInt16<512 >::Recurse(simdLlrs, SimdBitset<512>((const __m256i*)rawFrozenBits),
      messageLists, codewordLists, accumulatedMetric, decisionStage);
    break;
  case 1024:
    listOut = PolarListRecursiveInt16<1024 >::Recurse(simdLlrs, SimdBitset<1024>((const __m256i*)rawFrozenBits),
      messageLists, codewordLists, accumulatedMetric, decisionStage);
    break;

  default:
    throw std::runtime_error("Unhandled code word size: " + std::to_string(numLlrs));
  }
}
```

3.4 List Initialization

The MATLAB code around the `if (decisionStep == 0)… else` lines are used to initialize the list. Briefly, the code operates using a list depth of 8 at all times. For the first three non-frozen bit decisions, this special handling ensures that the list is filled with all 8 possible bit patterns and their metrics.
So, for the first non-frozen position, the following code executes:

```
        if (decisionStep == 0)
            s = [st(1:4); sf(5:8);];
            accumMetric = accumMetric + [mt(1:4); mf(5:8);];
            decisionStep = 1;
            list = 1:maxListDepth;
        elseif (decisionStep == 1)
```

This fills the candidate list with the bit patterns: `[st st st st sf sf sf sf]`, where `st` and `sf` are the “correct” and “incorrect” bit-decisions. The list does not yet need pruning, so the returned list seqence is [1,2,3,4,5,6,7,8].
For the second non-frozen position, the following code executes:

```
elseif (decisionStep == 1)
            s = [st(1:2); sf(3:4); st(5:6); sf(7:8);];
            accumMetric = accumMetric + [mt(1:2); mf(3:4); mt(5:6); mf(7:8);];
            decisionStep = 2;
            list = 1:maxListDepth;
        elseif (decisionStep == 2)
```

This fills the candidate list with the bit patterns: `[st st sf sf st st sf sf]`. The list does not yet need pruning, so the returned list seqence is [1,2,3,4,5,6,7,8].
For the third non-frozen position, the following code executes:

```
        elseif (decisionStep == 2)
            s = [st(1); sf(2); st(3); sf(4); st(5); sf(6); st(7); sf(8);];
            accumMetric = accumMetric + [mt(1); mf(2); mt(3); mf(4); mt(5); mf(6); mt(7); mf(8);];
            decisionStep = 3;
            list = 1:maxListDepth;
        else
```

This fills the candidate list with the bit patterns: `[st sf st sf st sf st sf]`. The list does not yet need pruning, so the returned list seqence is [1,2,3,4,5,6,7,8].
This 3-step procedure ensures that at the fourth bit decision, the list is primed with the values:

```
st st st st sf sf sf sf
st st sf sf st st sf sf
st sf st sf st sf st sf
```

This ensures that list pruning may commence with all eight possible 3-bit sequences as a starting point.

This procedure would need to be extended to 4 steps for list-16, etc.

Metrics are updated whenever an **sf** "symbol-false" value is inserted, as no metric growth occurs for a true decision, **st** or "symbol true".
This procedure is handled efficiently using blend instructions in C++:

```
__m256i mt[4];
mt[0] = _mm256_blend_epi32(mt_zero, mf, 0xF0);
mt[1] = _mm256_blend_epi32(mt_zero, mf, 0xCC);
mt[2] = _mm256_blend_epi32(mt_zero, mf, 0xAA);
mt[3] = mt_zero;

__m256i sT_t[4];
sT_t[0] = _mm256_blend_epi32(sT, sF, 0xF0);
sT_t[1] = _mm256_blend_epi32(sT, sF, 0xCC);
sT_t[2] = _mm256_blend_epi32(sT, sF, 0xAA);
sT_t[3] = sT;
sT = sT_t[ref_decision];
```

3.5 List Merging

In the MATLAB code, the list merging is performed in the line: `list = list1(list2);`
In this code, list is a length 8 vector that contains values in the range [1...8] (or [0...7] in C++). For example, if list contained the values [1, 1, 2, 2, 3, 4, 4, 5] then this means that the returned bit patterns were obtained from the previous recursion's list at positions [1,1,2,2,3,4,4,5]. This is described in the diagram below.

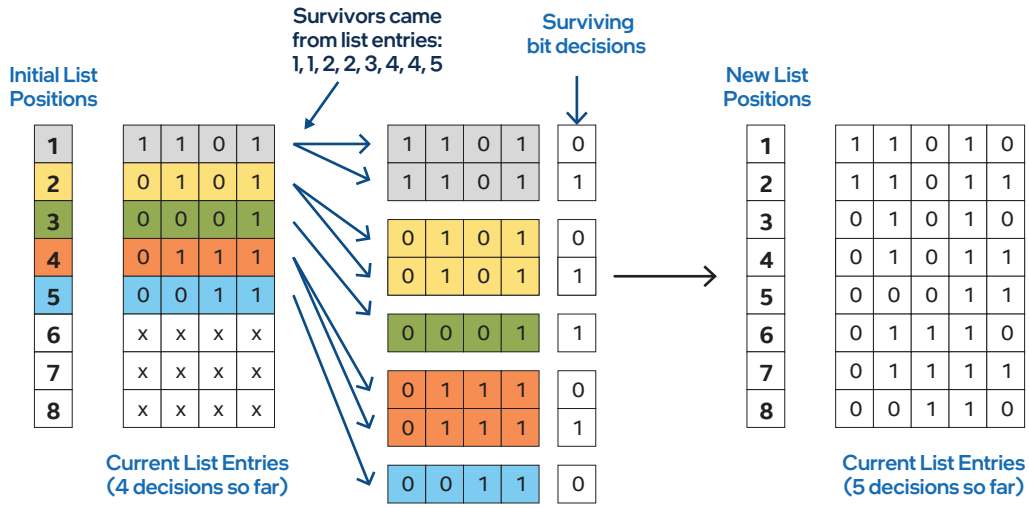

Figure 3-1: List Indexing

The MATLAB code above generates list1 for the left/upper leaves of the search tree and list2 for the right/lower branches of the search tree. When traversing back up the tree, these two lists must be merged. This is the purpose of the line: `list = list1(list2);`. In C++ this is handled by the function: MergeInPlaceInt16(`message`, list2.list, list1.msg_len);

This list merging procedure is illustrated in Figure 3-2.

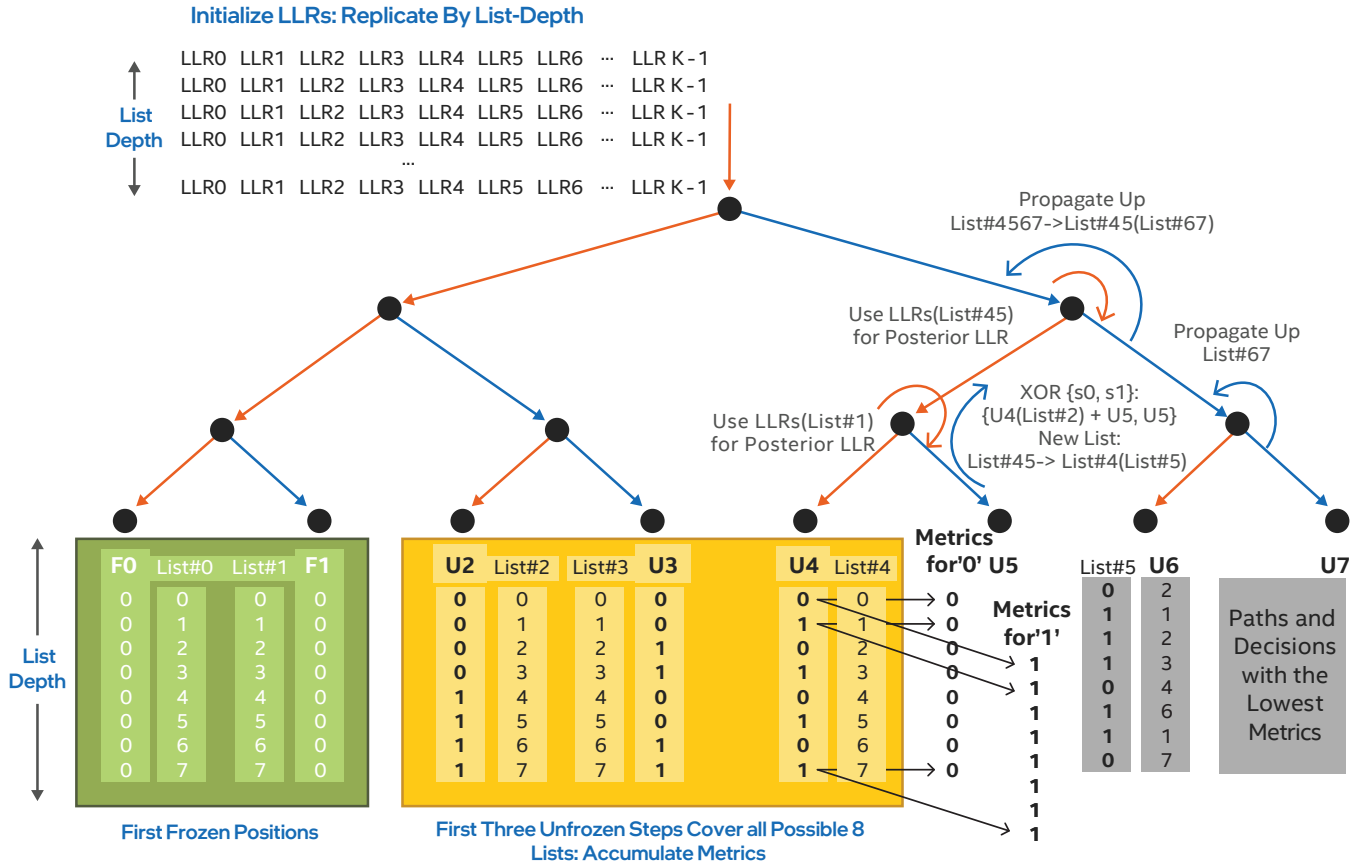

Figure 3-2: List Merging and Initial Decision Handling

This list merging procedure performs the following steps:

- List#4 (at leaf#U4 in Figure 3-2): [1,1,2,2,3,4,4,5]
- List#5 (at leaf#U5 in Figure 3-2): [1,2,2,3,4,5,5,6]
- Combined list propagated up: list#45 = `List4(list5)` = [List#4(1) List#4(2) List#4(2) List#4(3) List#4(4) List#4(5) List#4(5) list#4(6)] = [1 1 1 2 2 3 3 4]

The bit decisions are merged in the following MATLAB line: `s = [mod(s1(list2,:) + s2, 2) s2];` which is the C++ function **`XorInPlaceInt16<halfLlrs*8>(codeword, codeword + halfLlrs, list2.list);`**

3.6 List Pruning

Within the processing of all of the node types except Rate0 Nodes, multiple new candidates are produced for each input candidate. In each of these nodes, the candidates must be pruned back to the best 8 candidates (in list-8 processing).

The optimum method for doing this is to select the eight candidates with the lowest metrics. However, to do this requires a partial sorting and this adds an unacceptable number of instruction cycles to the decoding.

An alternative is to use a suboptimal mean. The MATLAB code is below:

```
%Metric prune function: mean threshold
function idx = integerMeanMetricPrune(metricsIn, varargin)
global maxListDepth;

if nargin==2
    meanDepth = varargin{1};
else
    meanDepth = [2:(2+maxListDepth)];
end
%Take the mean from position 2 to position 2+maxListDepth
%The lowest metric will always be the first one and the ones above this
%range will tend to be highly skewed. This way, the mean is closer to the
%median.
threshold = floor(mean(metricsIn(meanDepth)));
%threshold = mean(metricsIn(2:(1+maxListDepth))); This also works!
low_idx = find( metricsIn < threshold);
hi_idx = find( metricsIn >= threshold);

idx = [low_idx; hi_idx;];
idx = idx(1:maxListDepth);
```

This code returns the selected list indices that indicate which indices remain in the survivor list. The survivor list is, in fact, the value returned by this function modulo 8 (+1 for MATLAB indexing).

This code is represented by the function `PruneMetricsInt32(lowerMetric, upperMetric)` in C++. These functions take the mean of the metrics from metric positions [2,3,4,5,6,7,8,9,10] ([1,2,3,4,5,6,7,8,9] in C++ indexing). A justification for this is included in Figure 3-3, below.

Briefly, this method partially exploits the fact that the first 8 metrics will always be partially ordered. Values below the mean of these entries will likely be good candidates.

In the C++ code, it is necessary to take the array of metrics and iteratively search through the list to find all the values less than the mean and store their indexes in the survivor list. 3rd generation Intel® Xeon® processors, and beyond, will have additional Intel AVX-512 instructions that compress vectors. The compress instruction allows a single compare operation to be applied across the entire vector (i.e., is the element lower than the mean), returning a gap-less vector containing the element values that satisfied the test.

The `_mm512_mask_compress_epi16 (__m512i src, __mmask32 k, __m512i a)` intrinsic is well suited to this, and details of this instruction can be found in [3].

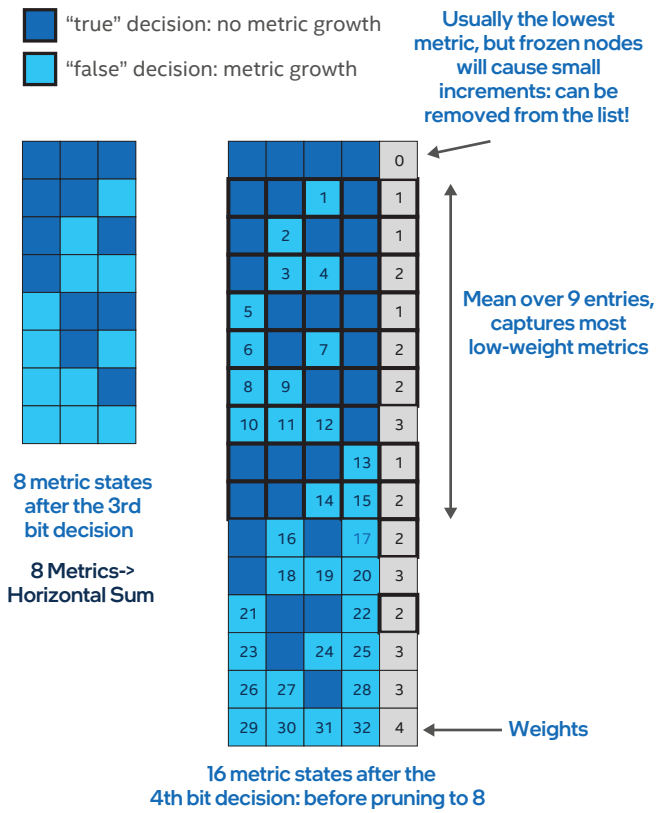

Figure 3-3: Justification of Mean Metric Pruning

3.7 Simplified Successive Cancellation Nodes

The C++ and MATLAB code can be modified so that the recursions stop at the node types identified in [4]. This so-called *Simplified Successive Cancellation List* decoder executes in fewer steps, as the recursions need not always progress to each leaf node. Instead, groups of leaf nodes may be processed in a near maximum likelihood way.

In the example below, the frozen nodes $u_0 \ldots u_6$ and unfrozen node u_7 match the pattern that is called a repetition node, while nodes $u_8 \ldots u_{15}$ match the pattern associated with a Single Parity Check Node.

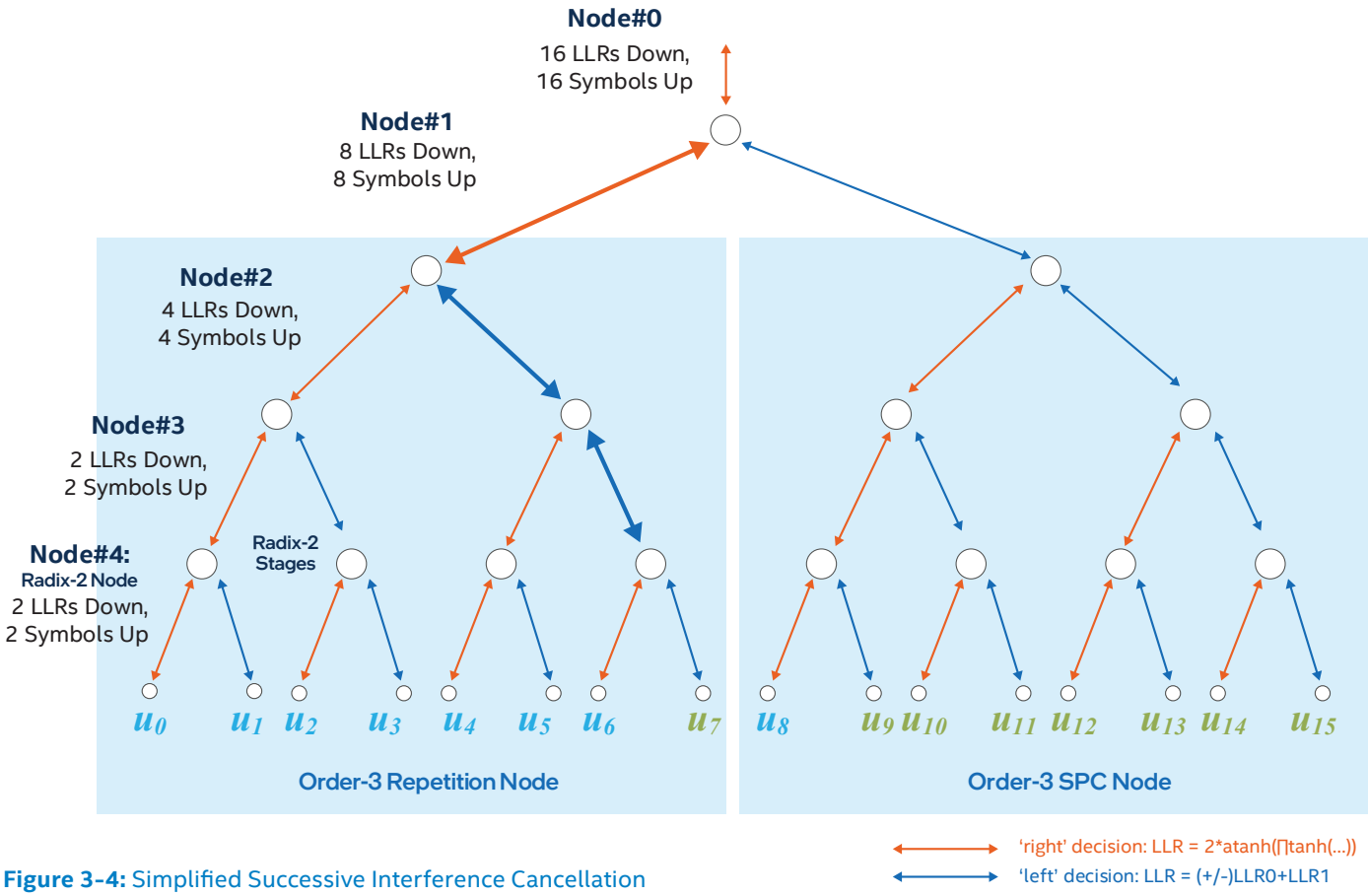

Figure 3-4: Simplified Successive Interference Cancellation

4 BLER vs SINR Performance Results

Execution time performance can be obtained from the FlexRAN SDK Performance report. However, execution time alone does not validate the usefulness of the decoder for 3GPP 5NGR applications. The probability of decoding error is the key metric for wireless performance. The FlexRAN SDK polar list decoder has been placed within a MATLAB MEX wrapper, allowing it to be used in MATLAB simulations.

The BLER performance was measured using this MATLAB model, and the probability of a block error was computed after at least 50 error events.

4.1 Example BLER-vs-SINR Curves for the SSC-List8-CRC decoder

Figure 4-1 shows the SINR vs BLER performance of the Simplified Successive Cancellation List-8 CRC-Aided decoder, implemented using the Intel AVX-512 instruction set.

The selection of the codeword size, k, and the message size (excluding the 3GPP length-11 CRC), E, was taken from the FlexRAN SDK version 20.08 unit tests.

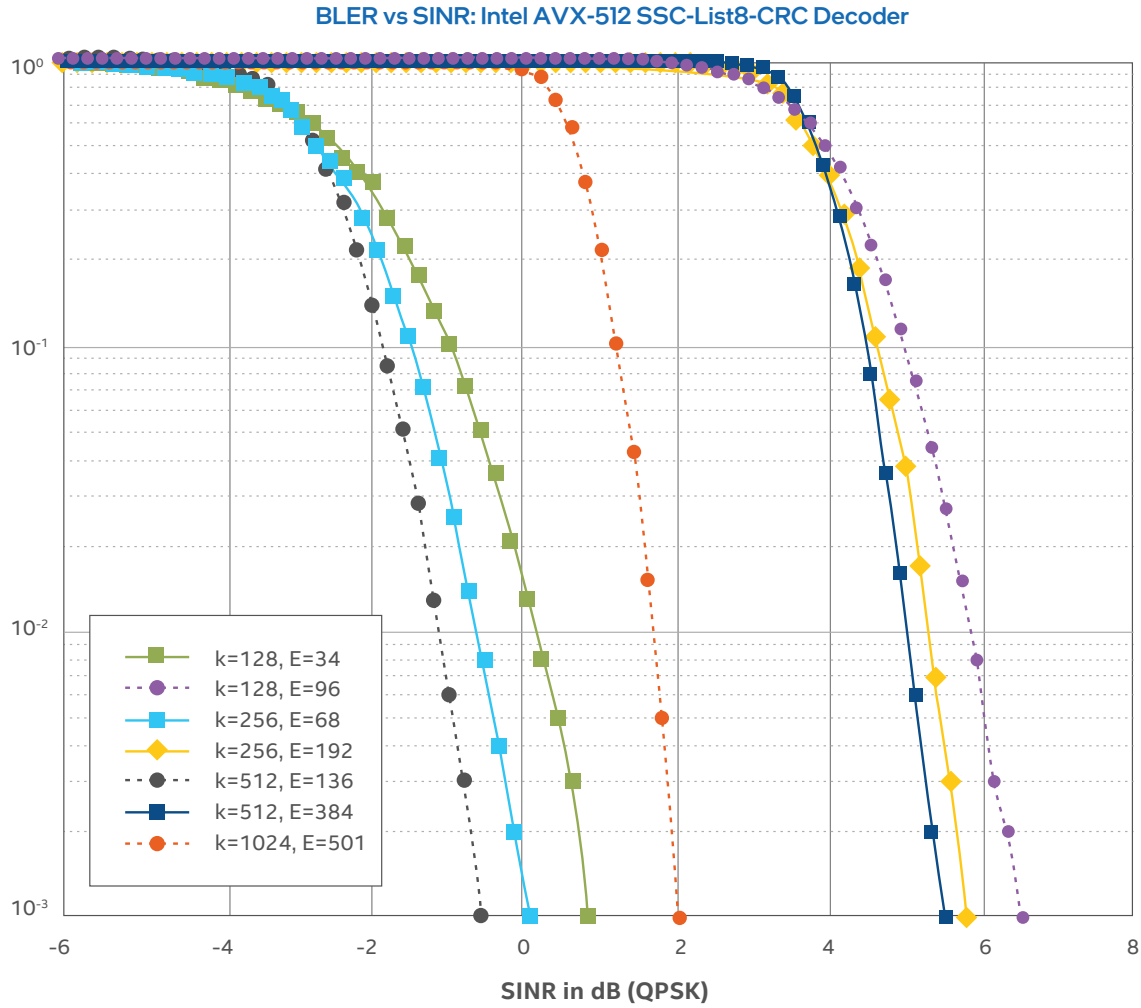

Figure 4 1: BLER-vs-SINR Performance of the SSC-List8-CRC Decoder (3GPP 5GNR Polar Construction)