An Efficient, Portable and Generic Library for Successive Cancellation Decoding of Polar Codes

Adrien Cassagne[1,2] Bertrand Le Gal[1] Camille Leroux[1]
Olivier Aumage[2] Denis Barthou[2]

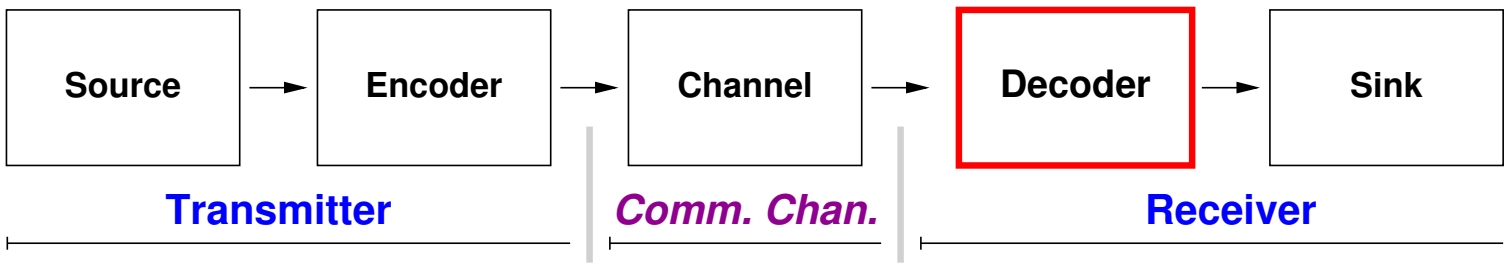
[1]IMS, Univ. Bordeaux, INP, France

[2]Inria / Labri, Univ. Bordeaux, INP, France

LCPC, September 2015

Context: Error Correction Codes (ECC)

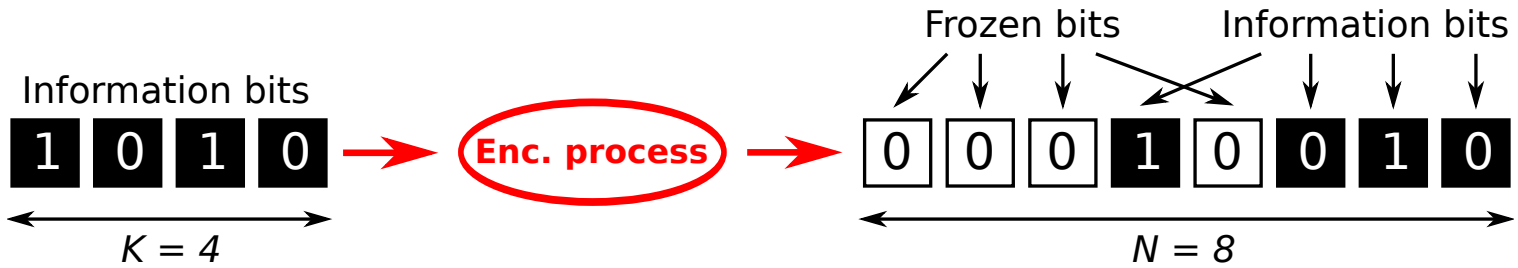
- Algorithm that enable reliable delivery of digital data
  - Redundancy for error correction
- Usually implemented in hardware
- Growing interest for software implementation
  - End-user utilization (low power consumption processors)
  - Less expensive production than dedicated hardware chips
  - Algorithms validation (typically Monte-Carlo HPC simulations)
- Require performance
- Focus on the decoder (most time consuming part)



The communication chain

Polar Codes as a New Class of ECC

- Explored for upcoming 5G Mobile Phones (Huawei[1])
- Redundancy: adding bits at fixed positions (value is always 0)



Example of Polar Code (number of info. bits $K = 4$, frame size $N = 8$)

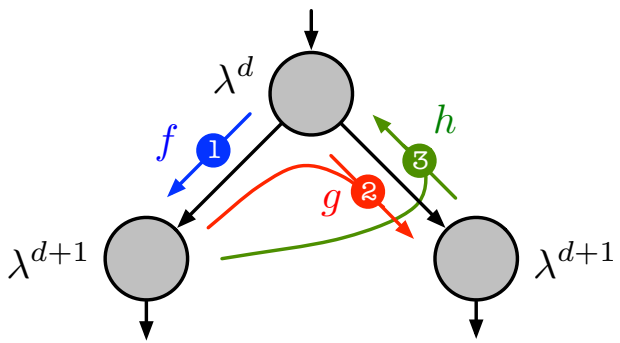
- Rate $R = N/K$: frame size / information bits ratio

[1] http://www.huawei.com/minisite/has2015/img/5g_radio_whitepaper.pdf

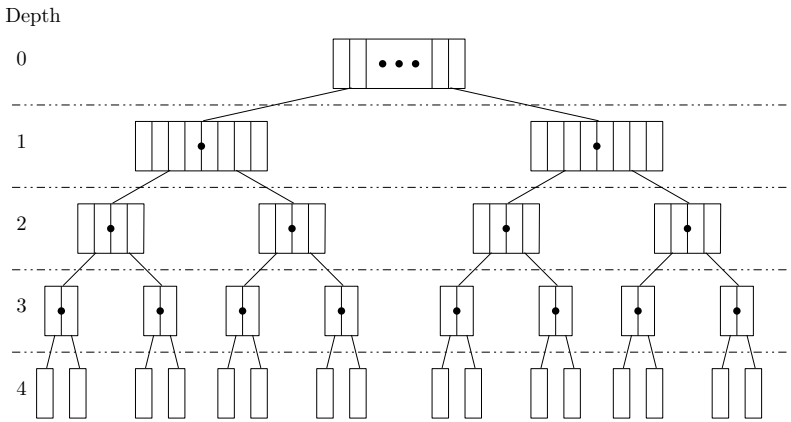
The Successive Cancellation (SC) Algorithm

- Depth-first binary tree traversal/search algorithm
- 3 key functions:

$$\begin{cases} \lambda_c & = & f(\lambda_a, \lambda_b) & = & sign(\lambda_a.\lambda_b).\min(|\lambda_a|, |\lambda_b|) \\ \lambda_c & = & g(\lambda_a, \lambda_b, s) & = & (1-2s)\lambda_a + \lambda_b \\ (s_c, s_d) & = & h(s_a, s_b) & = & (s_a \oplus s_b, s_b). \end{cases}$$

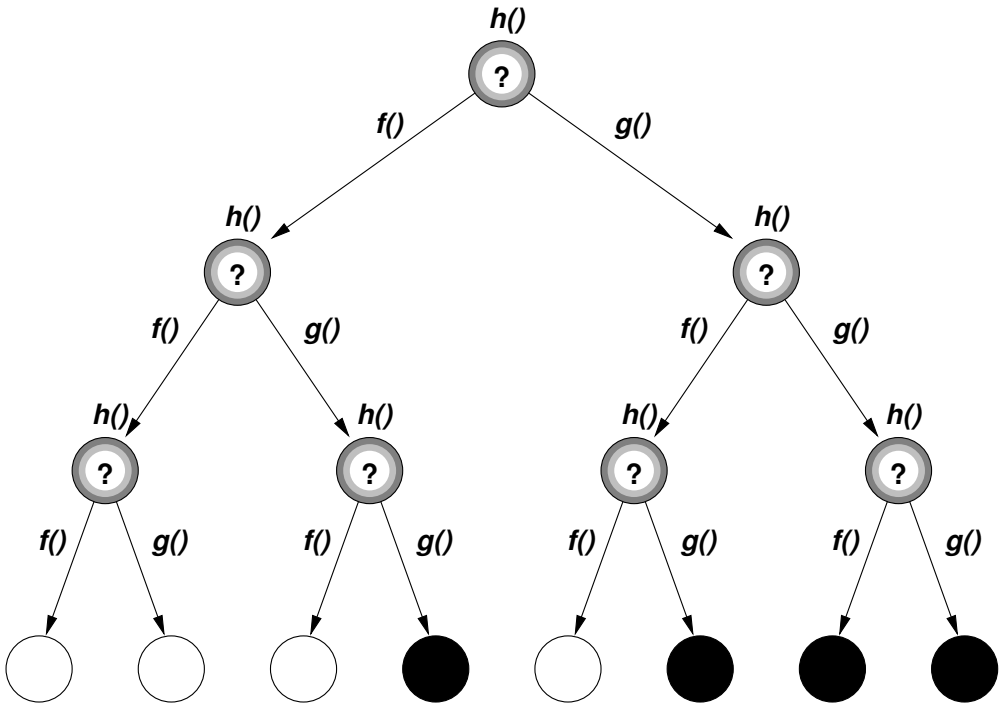


Per-node downward and upward computations



Data layout representation

Polar Decoding Tree



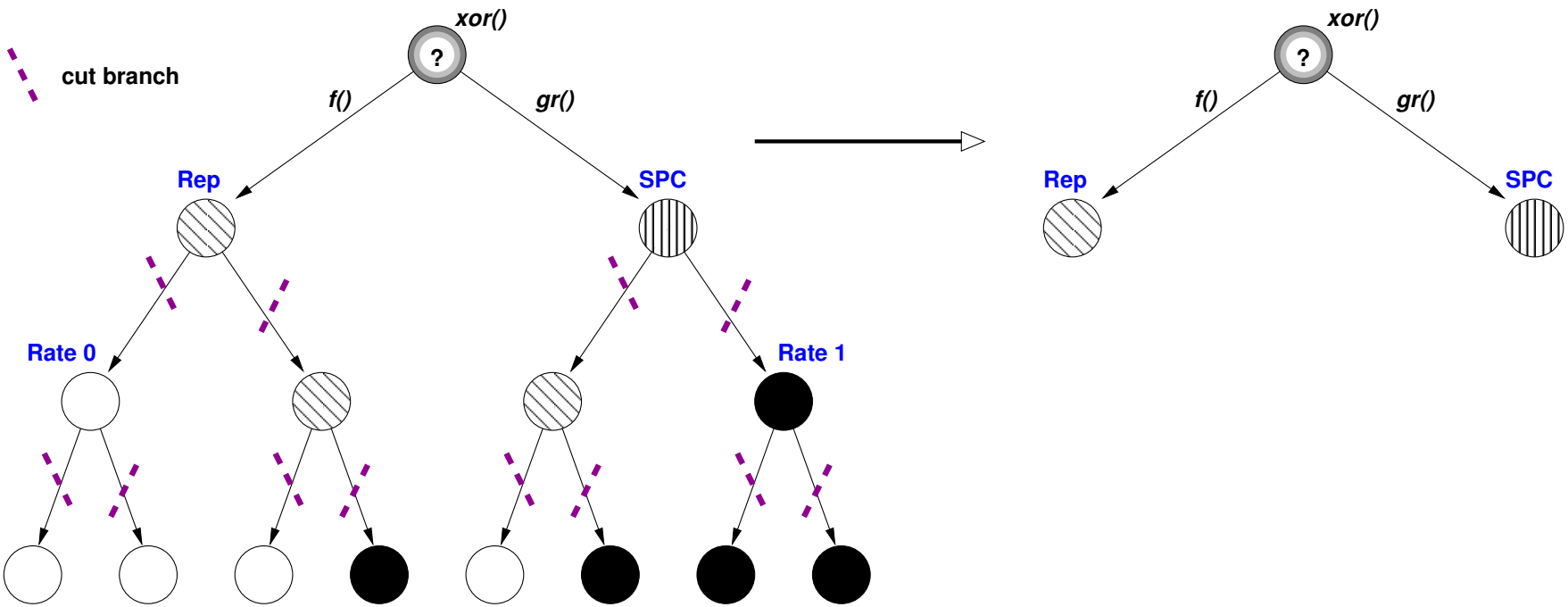
Position of the frozen and information bits in the tree

- Same specialized tree for each frame
- Frames are independent

A Wide Optimization Space

- Simplification of the computations (tree pruning, rewriting rules)
- Vectorization of the node functions (f, g, h)
- Optimization on the decoder binary size
- Implementation of low level kernels: various instruction sets (SSE, AVX, NEON, etc.)

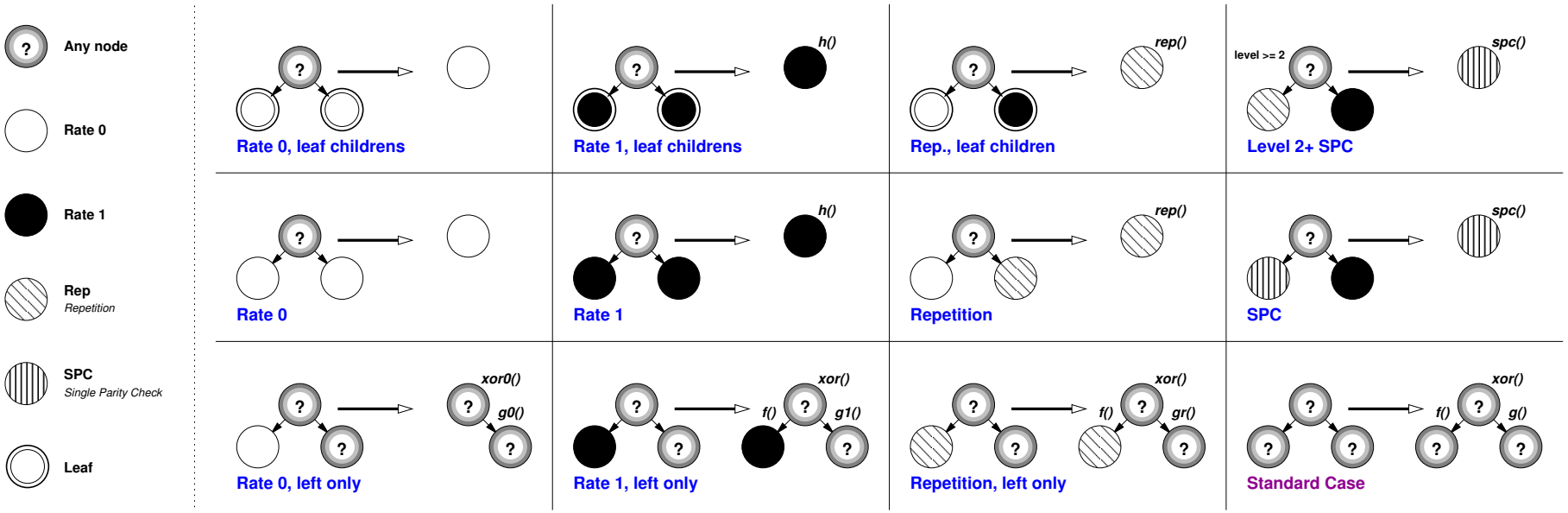
Example: Application of the Rewriting Rules



Rewriting rules applied to a $N = 8$ and $K = 4$ frame

- Rewriting rules are applied recursively
- Repeated application of this rules lead to a simplified tree

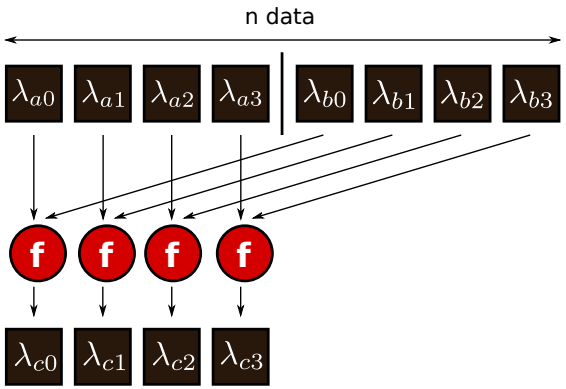
Rewriting Rules and Tree Pruning



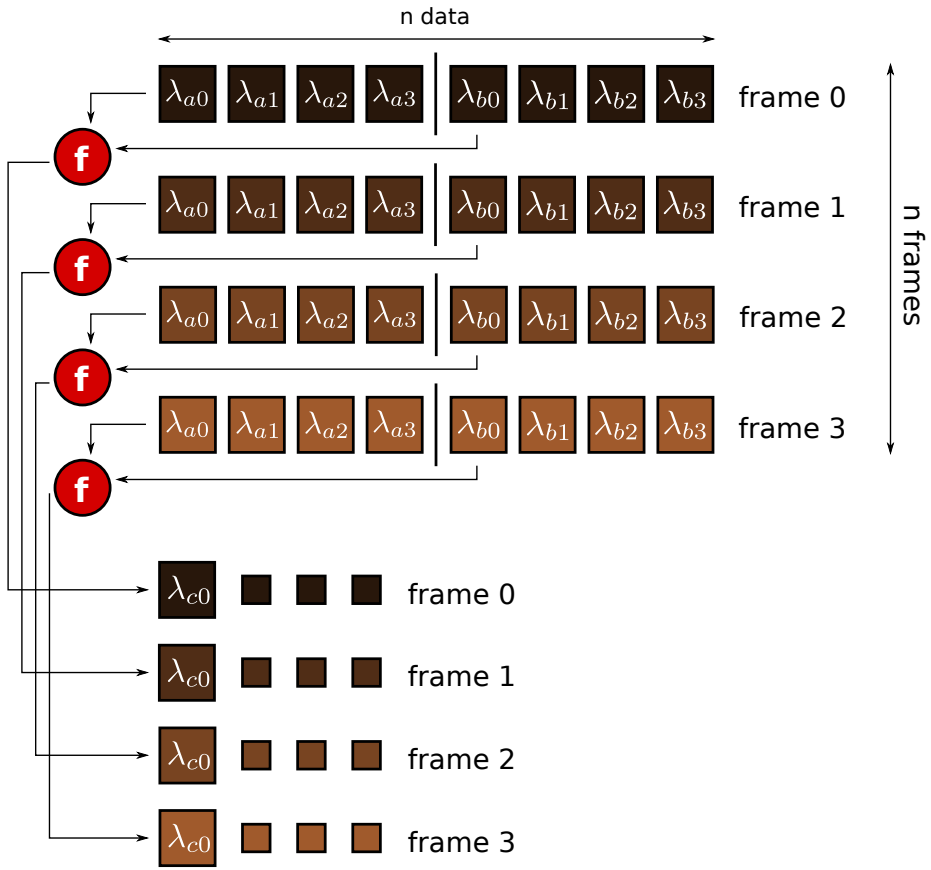
Sub-tree rewriting rules and tree pruning for processing specialization

Vectorization

Intra frame SIMD strategy:



Inter frame SIMD strategy:

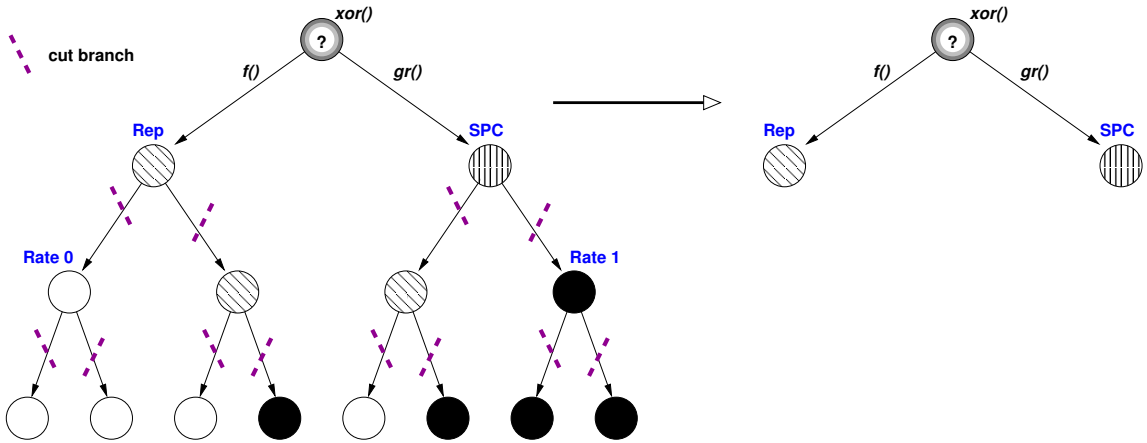


P-EDGE: a Dedicated Framework for Polar Codes

Features

1. Code generation
   - Flattening recursive calls
   - Rewriting rules
   - Generation of templated C++
2. C++ specialization
   - Loop unrolling
   - Data types
   - SIMD

P-EDGE: Code Generation Example



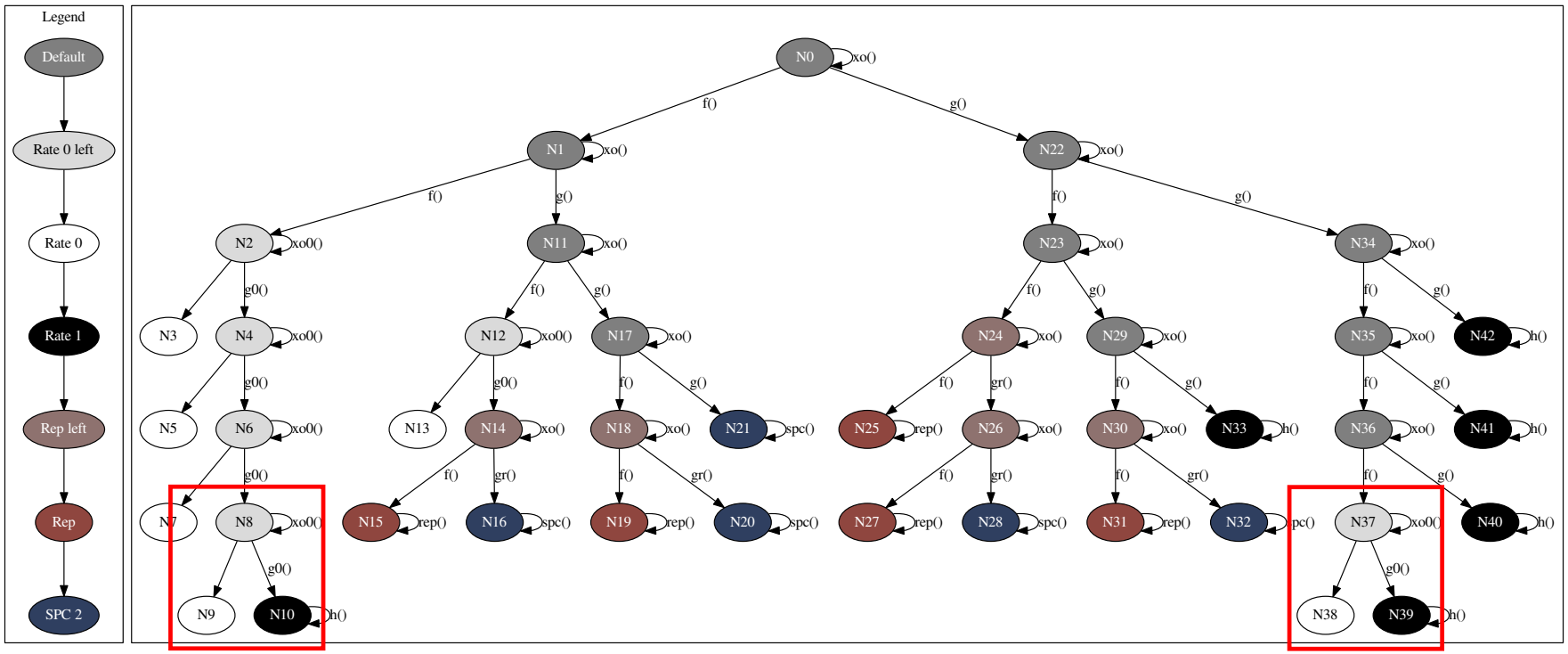
Generated code for $N = 8$ and $K = 4$

```
1  void Generated_SC_decoder_N8_K4::decode()
2  {
3    //   ------------template args---------------                -std args-
4    //   -types- --funcs-- ----offsets---- -size-                --buffs---
5    f   <     R , F , FI  , 0 , 4 ,     8 , 4    > :: apply (  l        );
6    rep < B , R , H , HI  , 8 ,         0 , 4    > :: apply (  l , s    );
7    gr  < B , R , G , GI  , 0 , 4 , 0 , 8 , 4    > :: apply (  l , s    );
8    spc < B , R , H , HI  , 8 ,         4 , 4    > :: apply (  l , s    );
9    xo  < B ,     X , XI  , 0 , 4 ,     0 , 4    > :: apply (      s    );
10 }
```

Reducing the L1I Cache Occupancy

- Flattening may generate large binaries
- Binary size grows with the frame size
- Performance slowdown when the binary exceeds the L1I cache
- Moving offsets from template to function arguments
  - Help the compiler to factorize many function calls

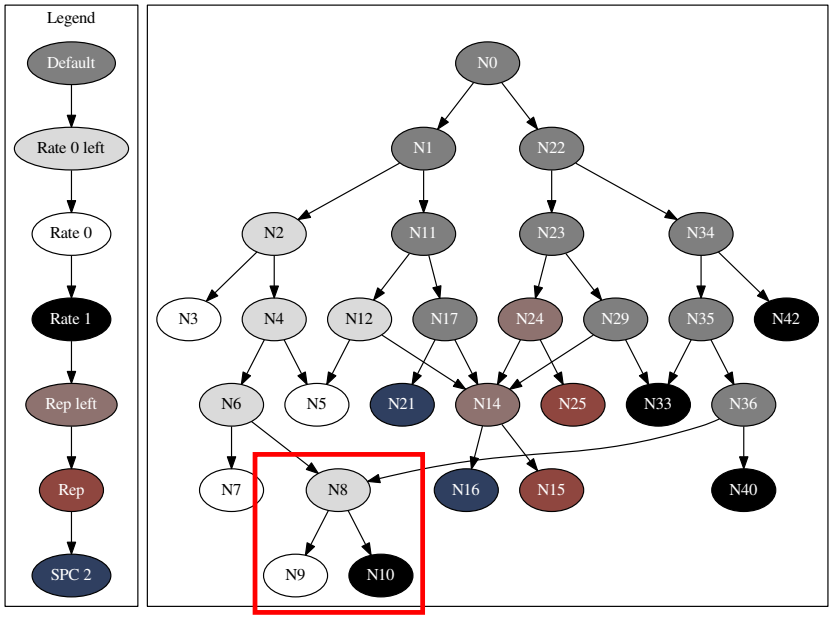
Sub-tree Folding Technique



Full decoding tree representation ($N = 128, K = 64$).

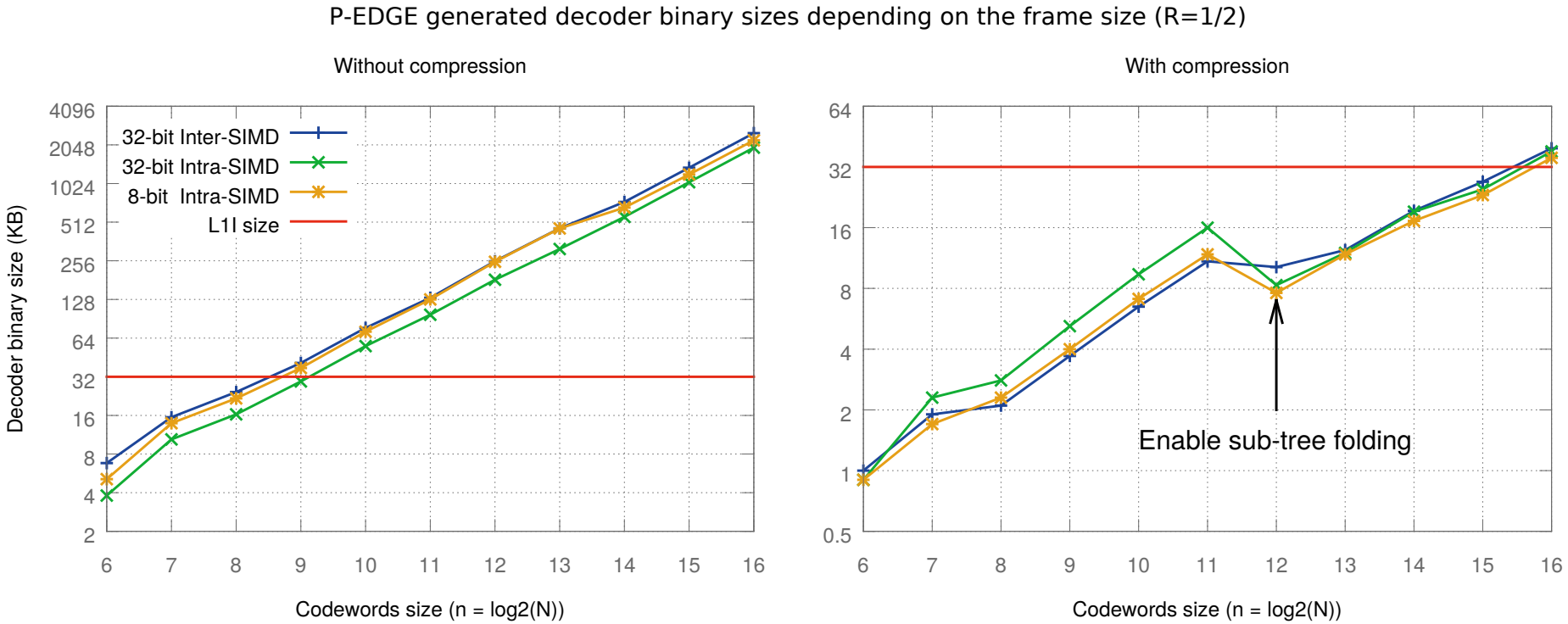
Sub-tree Folding Technique

Enabling compression



- A single occurrence of a given sub-tree traversal is generated, and reused wherever needed
- Compression ratio on the example shown: 1.48

Binary Sizes Comparison



- De-templetization: **10-fold** binary reduction
- Tree folding: **5-fold** binary reduction (for $N = 2^{16}$)

Performance results

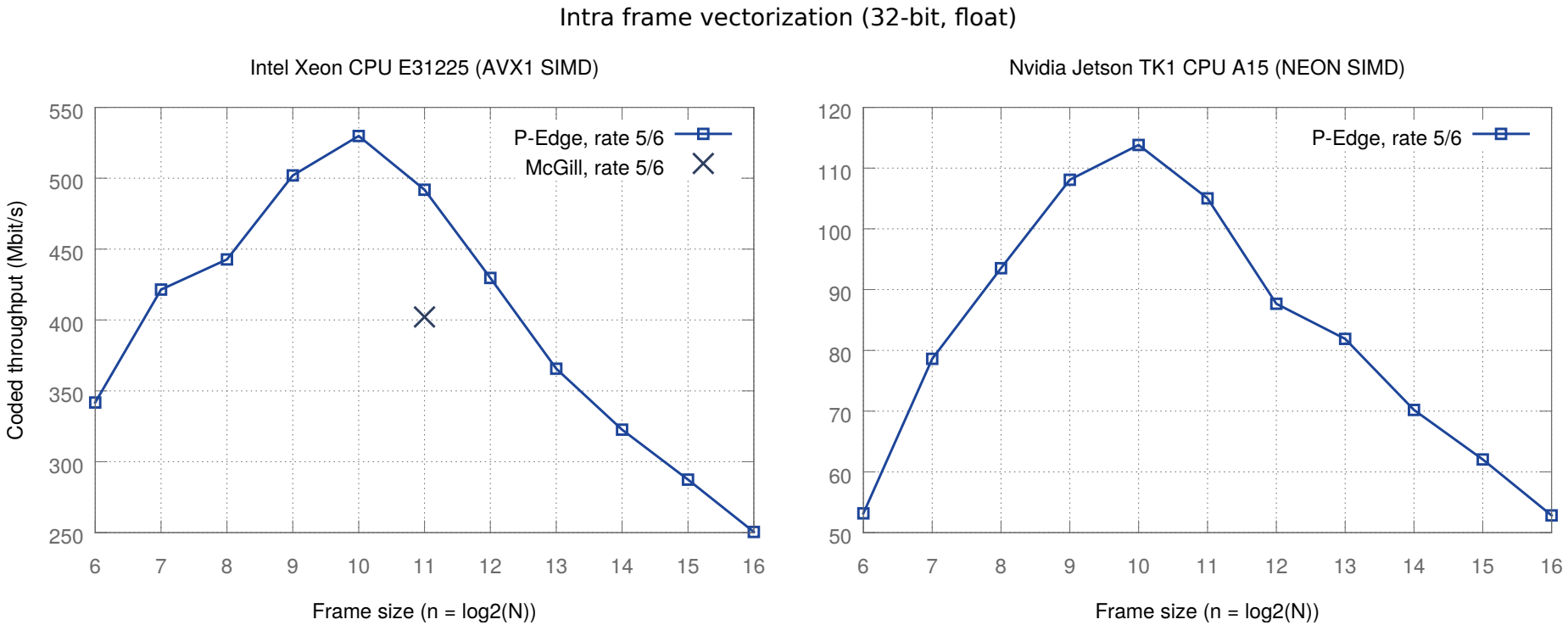
	P-Edge Intel-based	P-Edge ARM-based	McGill impl.[2]
CPU	Intel Xeon E31225 3.10Ghz	ARM Cortex-A15 MPCore 2.32GHz	Intel Core i7-2600 3.40GHz
Cache	L1I/L1D 32KB L2 256KB L3 6MB	L1I/L1D 32KB L2 1024KB No L3	L1I/L1D 32KB L2 256KB L3 8MB
Compiler	GNU g++ 4.8	GNU g++ 4.8	GNU g++ 4.8

Performance evaluation platforms.

- Compiler flags: `-std=c++11 -Ofast -funroll-loops`

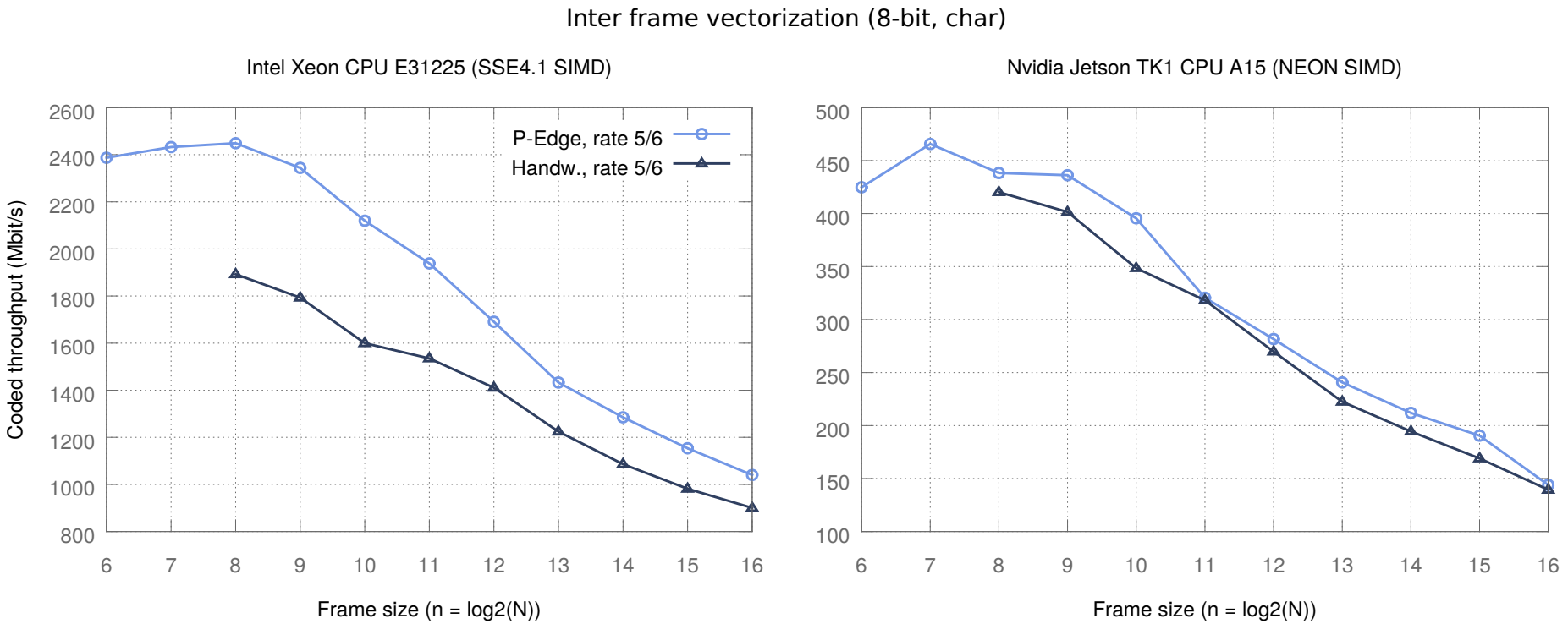
[2] G. Sarkis, P. Giard, C. Thibeault, and W.J. Gross. Autogenerating software polar decoders. In Signal and Information Processing (GlobalSIP), 2014 IEEE Global Conference on, pages 6–10, Dec 2014.

Intra-SIMD Comparison with State of Art



- Cross marks: Sarkis et al.
- **P-Edge up to 25% better**

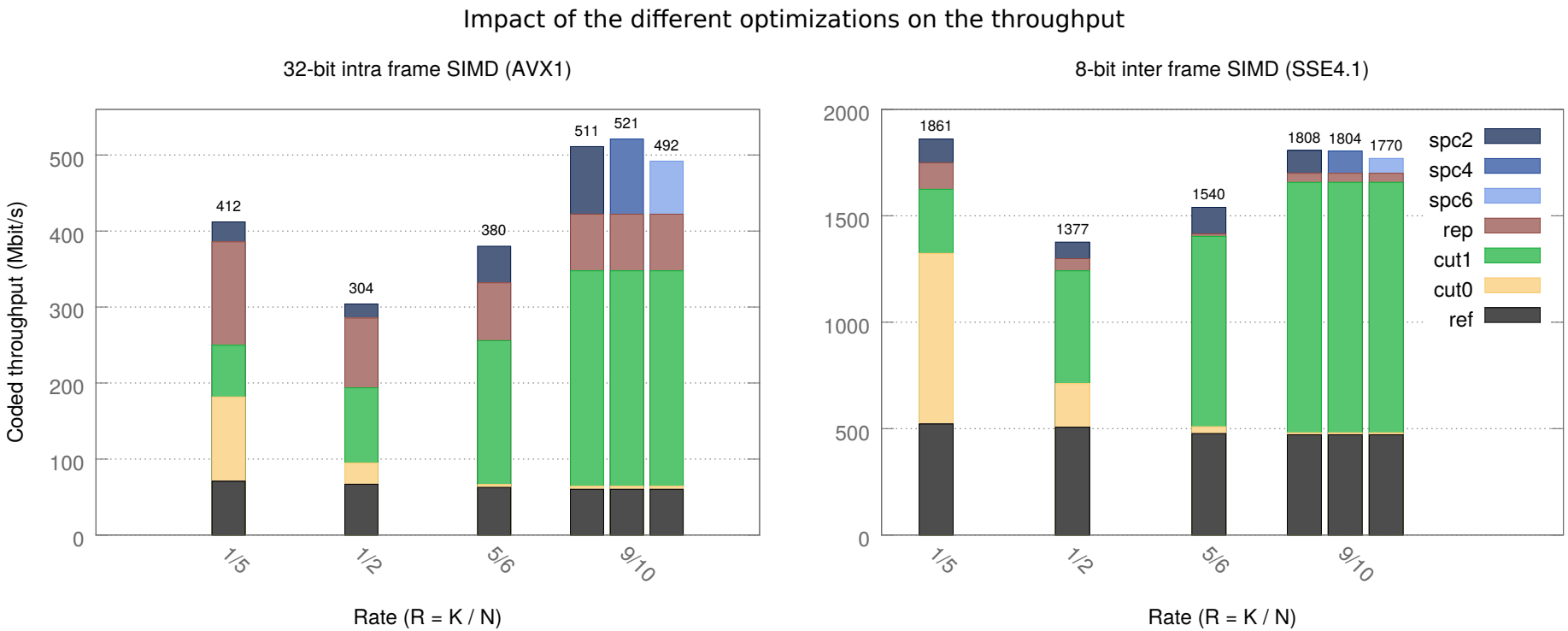
Inter-SIMD Comparison with State of Art



- Triangles marks: former "handwritten" implementation[3]
- **P-Edge up to 25% better**

[3] B. Le Gal, C. Leroux, and C. Jego. Multi-gb/s software decoding of polar codes. IEEE Transactions on Signal Processing, 63(2):349–359, Jan 2015.

Exploring Optimization Impacts



Throughput depending on the different optimizations for $N = 2048$

Conclusion and Future Works

Conclusion:

- P-EDGE: a Polar ECC decoder exploration framework
- Clear separation of concerns
  - Rewriting rules engine
  - Low level building blocks (f, g, h, rep, spc, ...)
- Large optimization exploration
- Outperform state of art decoders
- Performance Portability

Future works:

- In-depth performance analysis
  - Performance model (*Roof-line*, *Execution-Cache-Memory* (ECM))
- Reduce memory footprint
- Explore other Polar code decoder variants