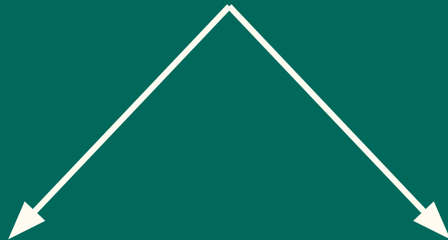
Оптимизации для протокола Cascade

—

Полина Киселева

Protocol	Block sizes (approx.) k_1	k_2	k_i	`Cascade` passes	`BICONF`	Block reuse	Shuffling	Singl. blocks
orig. Ref. [6]	$0.73/Q$	$2k_1$	$2k_{i-1}$	4	no	no	random	no
mod. (1) Ref. [10]	$0.92/Q$	$3k_1$	–	2	yes	no	random	no
opt. (2) Ref. [19]	$0.8/Q$	$5k_1$	$n/2$	10	no	no[a]	random	no
opt. (3)	$1/Q$	$2k_1$	$n/2$	16	no	no	random	no
opt. (4)	$1/Q$	$2k_1$	$n/2$	16	no	yes	random	no
opt. (5)	$1/Q$	$2k_1$	$n/2$	16	no	yes	determ.	no
opt. (6)	$1/Q$	$2k_1$	$n/2$	16	no	yes	random	yes
opt. (7)	$2^{\lceil \log_2 1/Q \rceil}$	$4k_1$	$n/2$	14	no	yes	random	no
opt. (8)	$2^{\lceil \alpha \rceil}$	$2^{\lceil (\alpha+12)/2 \rceil}$	$n/2^b$	14	no	yes	random	no

Оптимизации



Изменение алгоритма

Изменение параметров

(4)

(3)

Оптимизация 3

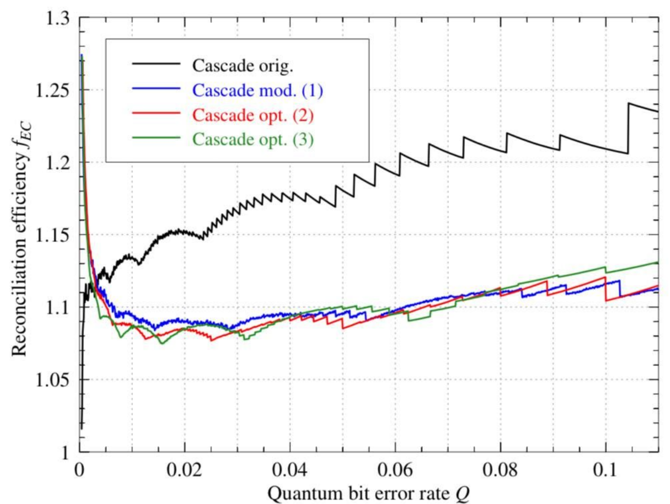
- $k_1 = 1/Q$ □ 1 ошибка на 1 блок
- $k_2 = 2k_1$ □ каскадный эффект компенсирует ошибки и с первого прохода
- $k_i = n/2$ □ BER достаточно низкий, чтобы разделить кадр на 2 блока
- 16 проходов для достижения низкого FER при больших блоках

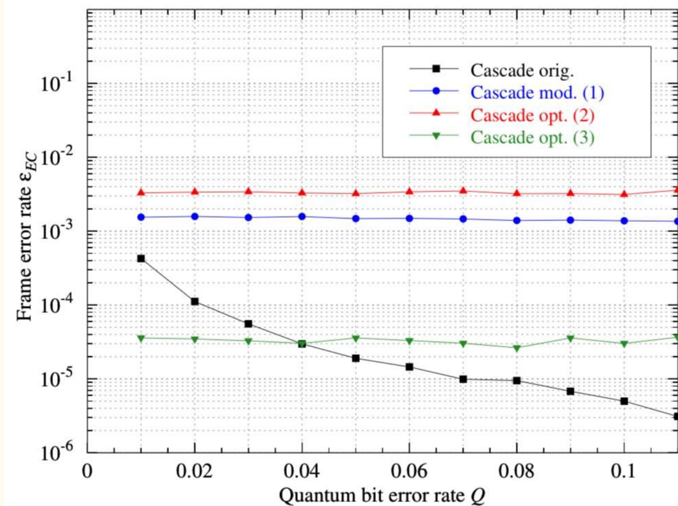
Оптимизация 3

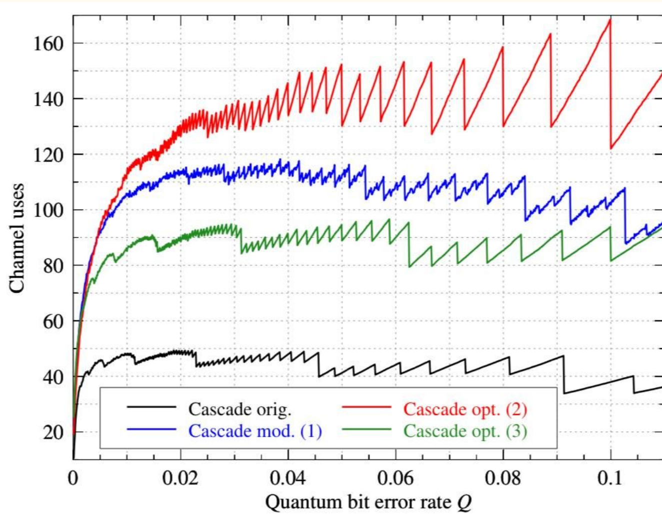
Почему лучше других?

- Наименьшее количество использований канала связи несмотря на 16 проходов (из-за больших блоков)
- Маленький FER: гарантия обнаружения всех ошибок
- Эффективность: меньше избыточность раскрытой информации

Сравнение результатов







Block reuse

- Храним указатели всех обработанных блоков
- Накопление и повторное использование уже проделанной работы
- “Экономим” вычисления четностей: раскрываем меньше, чем в стандартном Cascade

Без оптимизации

Пусть обнаружили ошибку на 4 проходе → каскадный процесс возвращается к проходу 1 → ошибка была в блоке, который уже проверили → заново проводим поиск по всему блоку, снова раскрываются все четности

VS

С оптимизацией

Пусть обнаружили ошибку на 4 проходе → каскадный процесс перебирает предыдущие блоки → находит запись о блоке, где был ошибочный бит → считает четность только у него

Block reuse

Было

- После исправления ошибки алгоритм переходит на следующий блок/проход, ошибки в других остаются
- Не хранит историю блоков
- Каждый раз вычисляем четности для всех блоков

Стало

- После исправления ошибки автоматически проверяются все блоки из истории, содержавшие этот бит
- Хранит историю блоков
- Раскрываем меньше четностей: только у тех блоков, где был ошибочный бит

Другие оптимизации

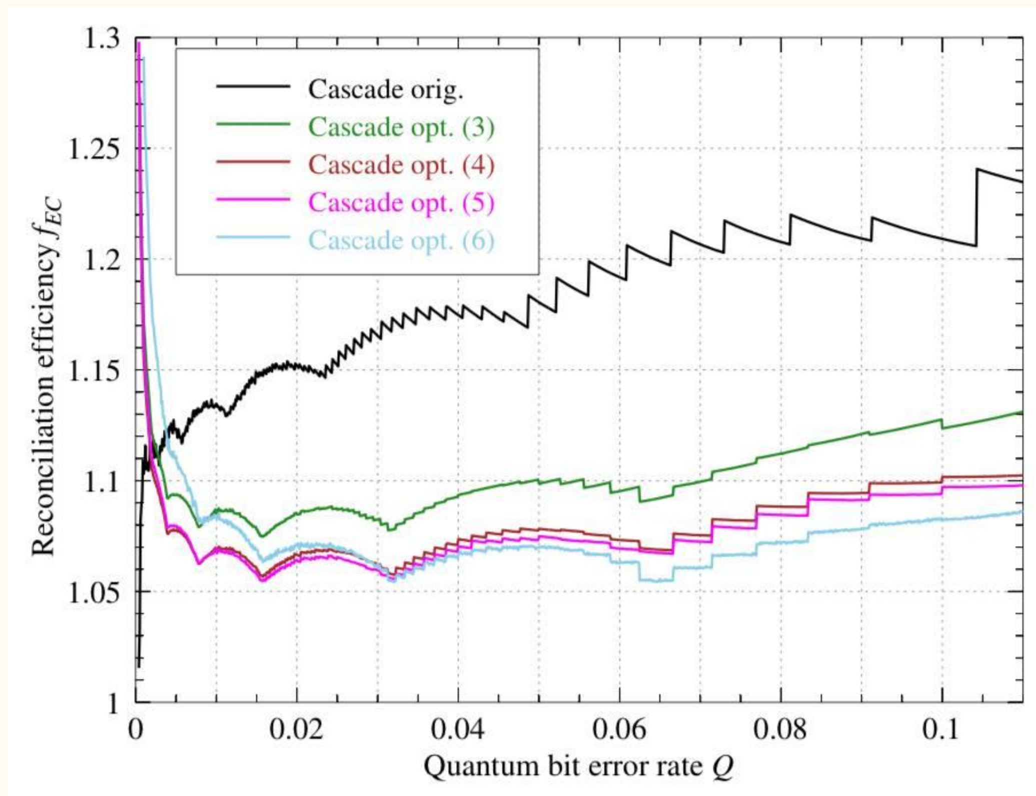
Deterministic Shuffling

Неслучайное перемешивание битов: используем конкретные алгоритмы перемешивания, чтобы гарантировать, что два бита из одного блока в проходе i никогда не окажутся в одном блоке в проходе i+1. Это нужно, чтобы минимизировать количество блоков, где четное число ошибок

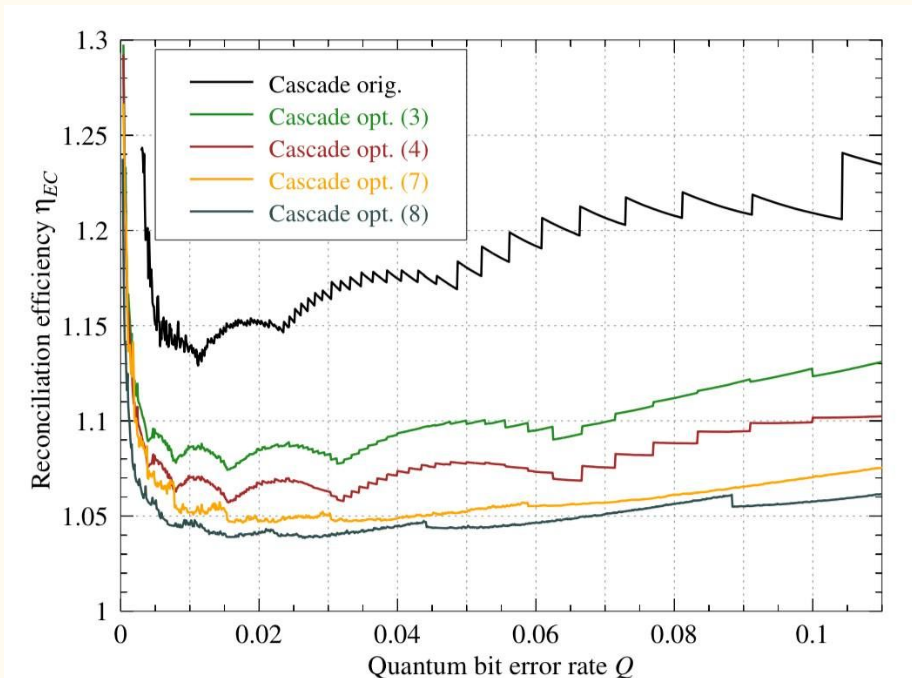
Singleton blocks

Блоки-одиночки - подблоки размера 1 с известным значением. Информация о них изымается в отдельный список, который потом используется для проверки напрямую. То есть при каждом разбиении блока у нас появляются синглтоны, информацию о которых не нужно считать и раскрывать заново

Сравнение результатов



Сравнение результатов



Оптимизации (3) и (4) обеспечивают хороший баланс между эффективностью, уровнем ошибок кадров и затратами на коммуникацию. Оптимизация (4) несколько эффективнее (3) при том же уровне ошибок кадров и затратах на коммуникацию, но её реализация сложнее и требует больше аппаратных ресурсов.