Ring Buffers As Fast As Possible

Content

- Motivation
- Queue Basics
- Interface
- Implementation
- Benchmarks

Terminology

- Single Producer/Consumer Bounded Queue
  - That’s a mouthful
- Queue or Ring Buffer will have to suffice
  - Queue: Fixed size elements
  - Ring Buffer: Variable size elements
- I will try to keep to this; If confused, please ask

Motivation

- Logging in real time environments
  - One Queue per thread
  - One thread emptying Queues
- Started benchmarking Queues

Queue Basics

- Element: Fundamental unit inside queue
  - Think: Node in a Linked List
- Produce: Adding elements to the queue
- Consume: Removing elements from the queue

Queue Basics – Members

- Storage
- Produce Position / Tail
- Consume Position / Head

Queue Basics – Produce

```
bool produce(const T& data) {
    auto next = (tail + 1) % SIZE;
    if (next == head) return false;
    buffer[tail] = data;
    tail = next;
    return true;
}
```

Queue Basics – Consume

```
bool consume(T& out) {
    if (head == tail)
        return false;
    out = buffer[head];
    head = (head + 1) % SIZE;
    return true;
}
```

Queue Interface – Produce

- Basic idea: No copies

  Not this:

  ```
  bool produce(const T&);
  bool produce(T&&);
  ```

- Forces copy or move
  - Instance of `T` needs to exist before call

Queue Interface – Produce

- Basic idea: No copies

  Solution: Emplace

  ```
  template<typename… Args>
  bool produce(Args&&…);
  ```

- Downside: Generates more code

Queue Interface – Produce

- Basic idea: No copies

  Alternative: Callback

```
template<typename Callback>
bool produce(Callback);
```

- Requires users to know about placement new
  - No publicly available Queue uses this

Queue Interface – Consume

- Basic idea: No copies

  Not this:

  ```
  bool consume(T&);
  std::optional<T> consume();
  ```

- Forces copy or move

Queue Interface – Consume

- Basic idea: No copies

  Solution candidate:

  ```
  T* peek();
  void consume();
  ```

- `peek` returns `nullptr` when empty
- `consume` must not be called if queue empty

Queue Interface – Consume

- Basic idea: No copies

  Solution: Callback

  ```
  template<typename Callback>
  bool consume(Callback);
  ```

- Downside: Generates more code

Queue Interface

- Pattern:
  1. Queue code – checks, setup
  2. User code – payload
  3. Queue code – commit

Interface Optimization

- Fixed vs. variable size of elements
  - Variable element size requires overhead in buffer
  - Could not find public implementation

Ring Buffer Interface

- Basic idea: No copies

  Not this:

  ```
  bool produce(void*, size_t);
  ```

Ring Buffer Interface – First Attempt

- Basic idea: No copies

  More like this:

  ```
  void* produce(size_t);
  ```

  Bugs abound

Ring Buffer Interface – Attempt #2

- Basic idea: No copies

```
void* produce_start(size_t);
void  produce_abort(size_t);
void  produce_commit(size_t);
```

- Hard to use correctly

Ring Buffer Interface – Attempt #3

- Basic idea: No copies

```
transaction produce_start(size_t);
void produce_abort(transaction&&);
void produce_commit(transaction&&);
```

- Still really hard to use correctly

Ring Buffer Interface

- Basic idea: No copies

```cpp
template<typename Callable>
bool produce(size_t, Callable);
```

- Least bad option, requires use of placement-new

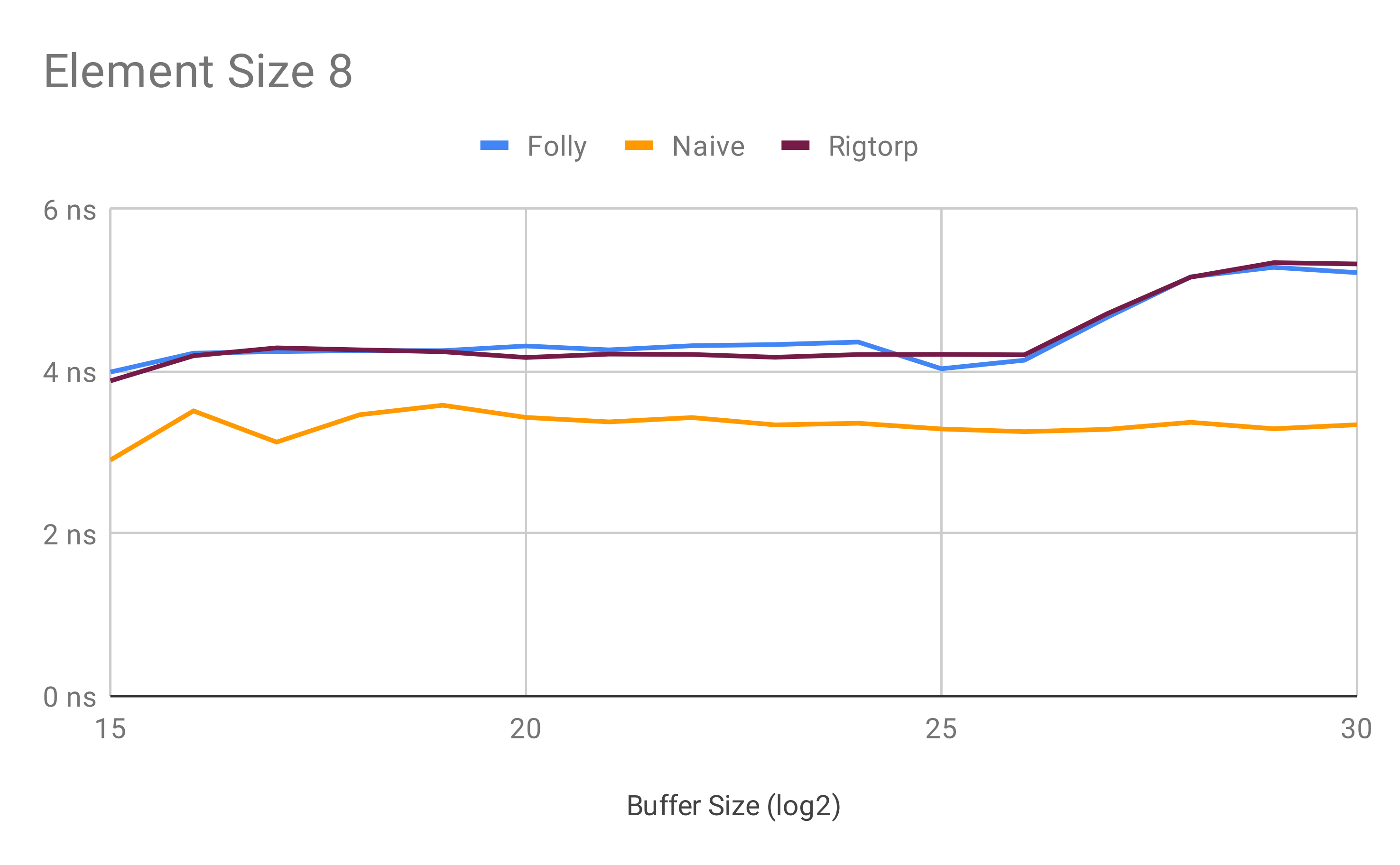
Possible Trade-Offs

- Arbitrary Buffer Size vs. Powers of Two
  - Low level optimization (modulo vs. bit-wise and)
  - Affects complexity of implementation

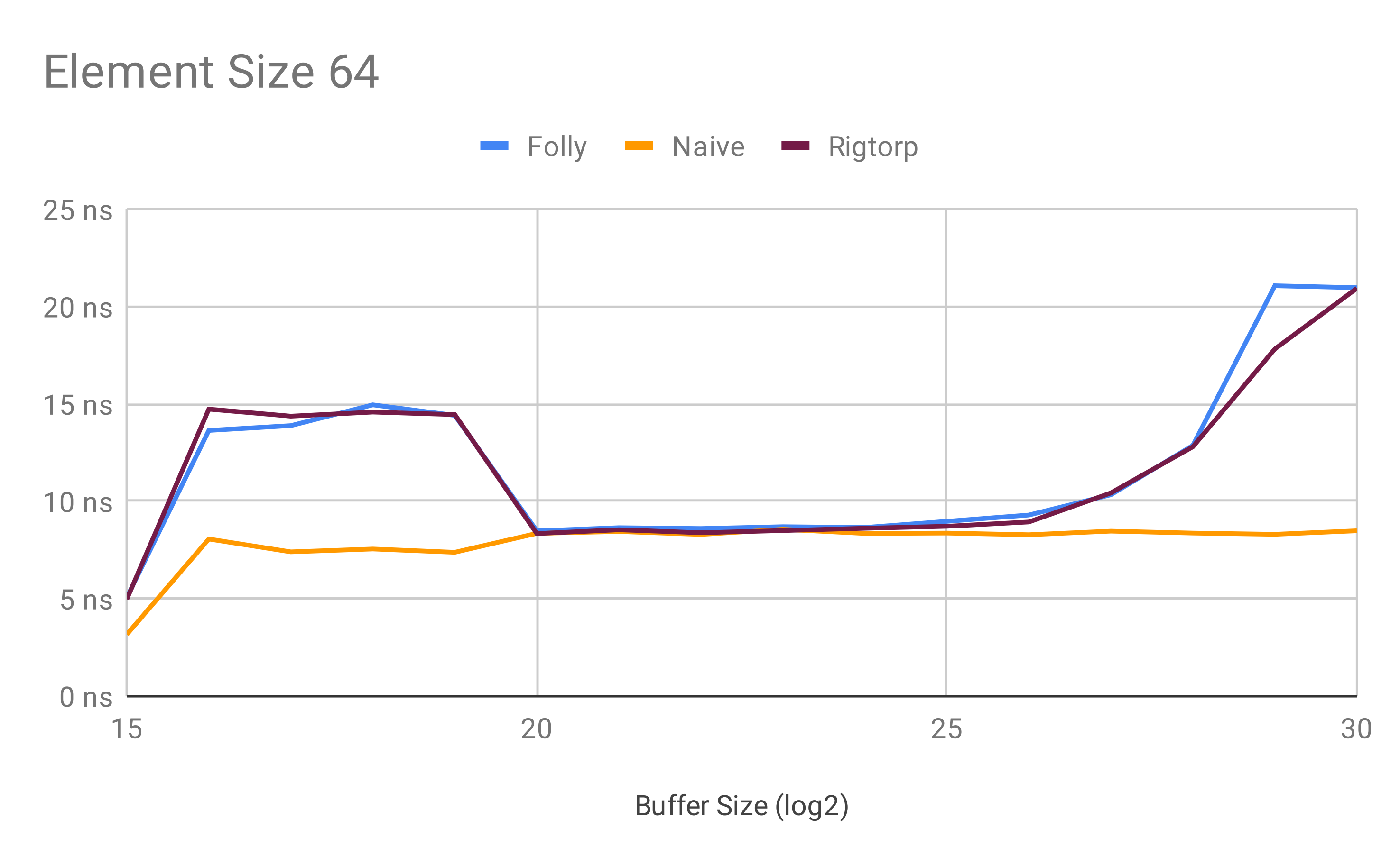
Possible Trade-Offs

- In-line Buffer vs. Heap-allocated Buffer
  - In-line Buffer cannot change size
  - Heap-allocated supports large sizes on MSVC
    - In-line only goes up to 2^31 - 1 Bytes
  - Additional indirection

In-Line vs. Heap – Benchmark



In-Line vs. Heap – Benchmark



Atomics

- Use Atomics with Acquire/Release Ordering

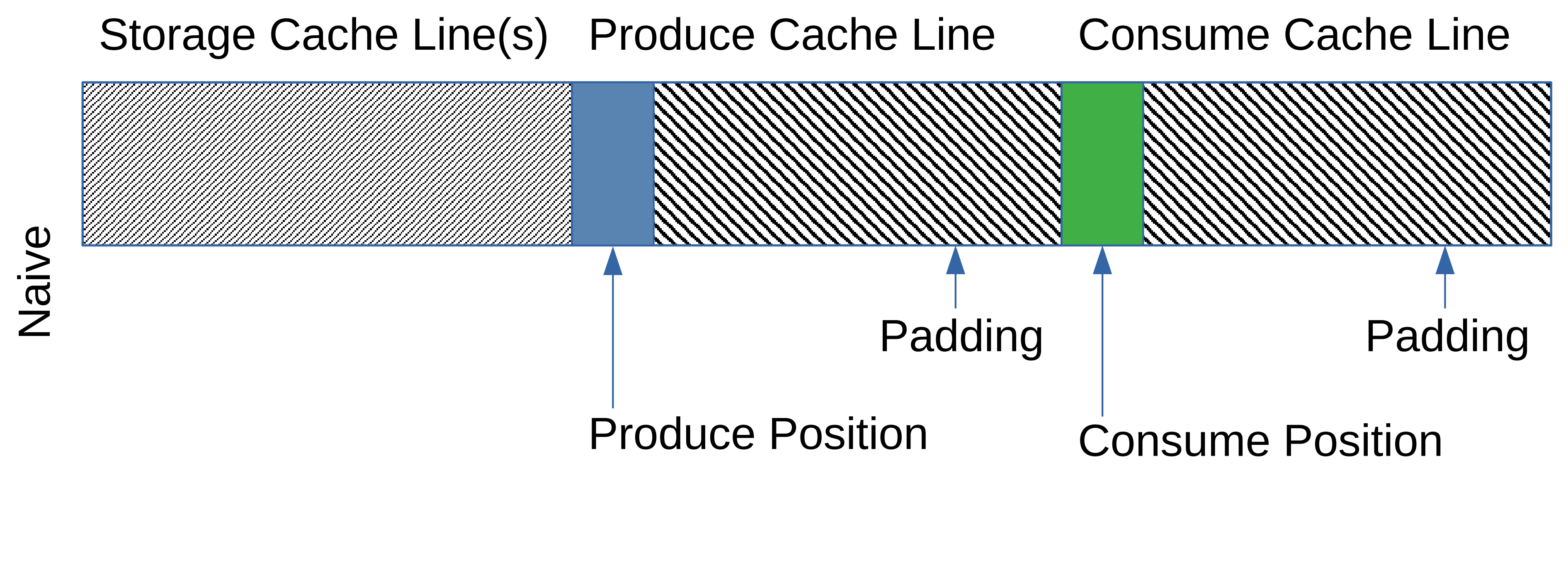
  All stores before a Release store will be visible in another thread after an Acquire load on the same atomic.

- Acquire/Release has no overhead on x86
  - Overhead exists on ARM

Prevent False Sharing

- Put Produce/Consume position each on their own cache line
- Single cache line is not enough to avoid false sharing for modern x86/x64 processors
  - Solution: two cache lines padding

Padded Layout



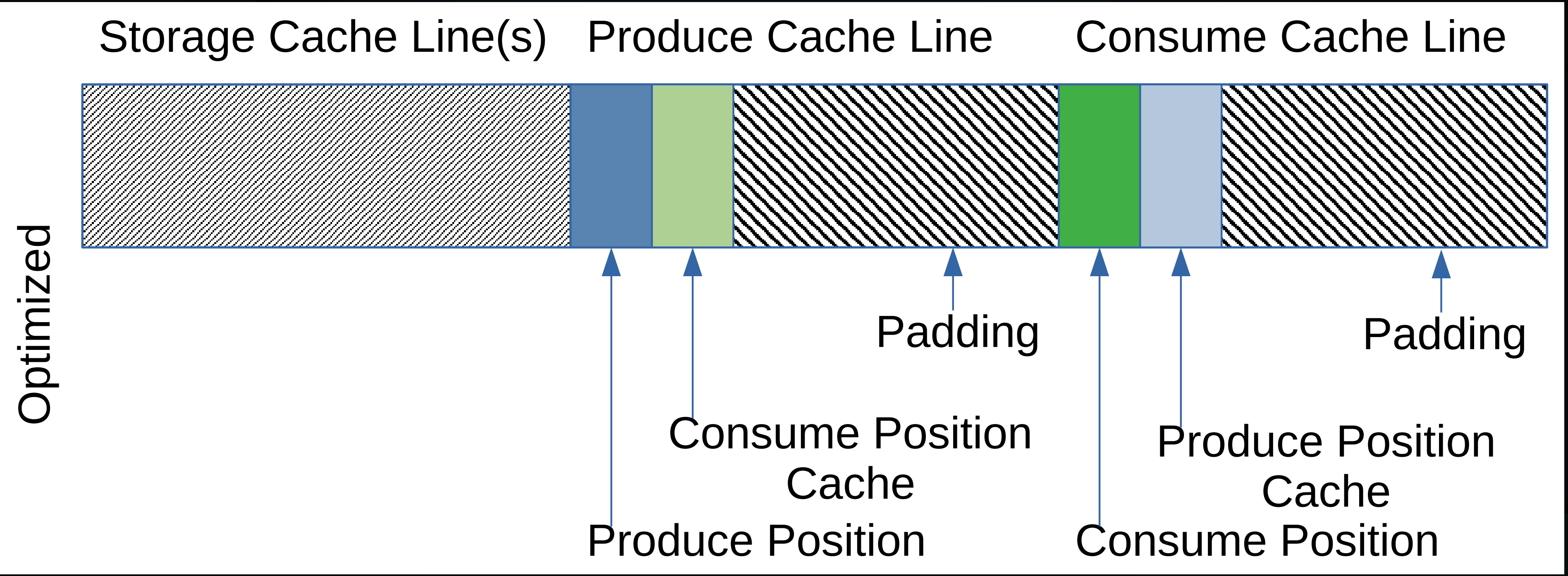
Optimization

- This is the common implementation
  - e.g.: folly, boost, rigtorp
- Straightforward to implement

Caching

- Every operation needs both positions
  - Constant synchronization needed between threads
- Solution: Cache Produce/Consume position
  - Cache for Consume in same cache line as Produce etc.
  - Only need to load one cache line if buffer is always empty/full

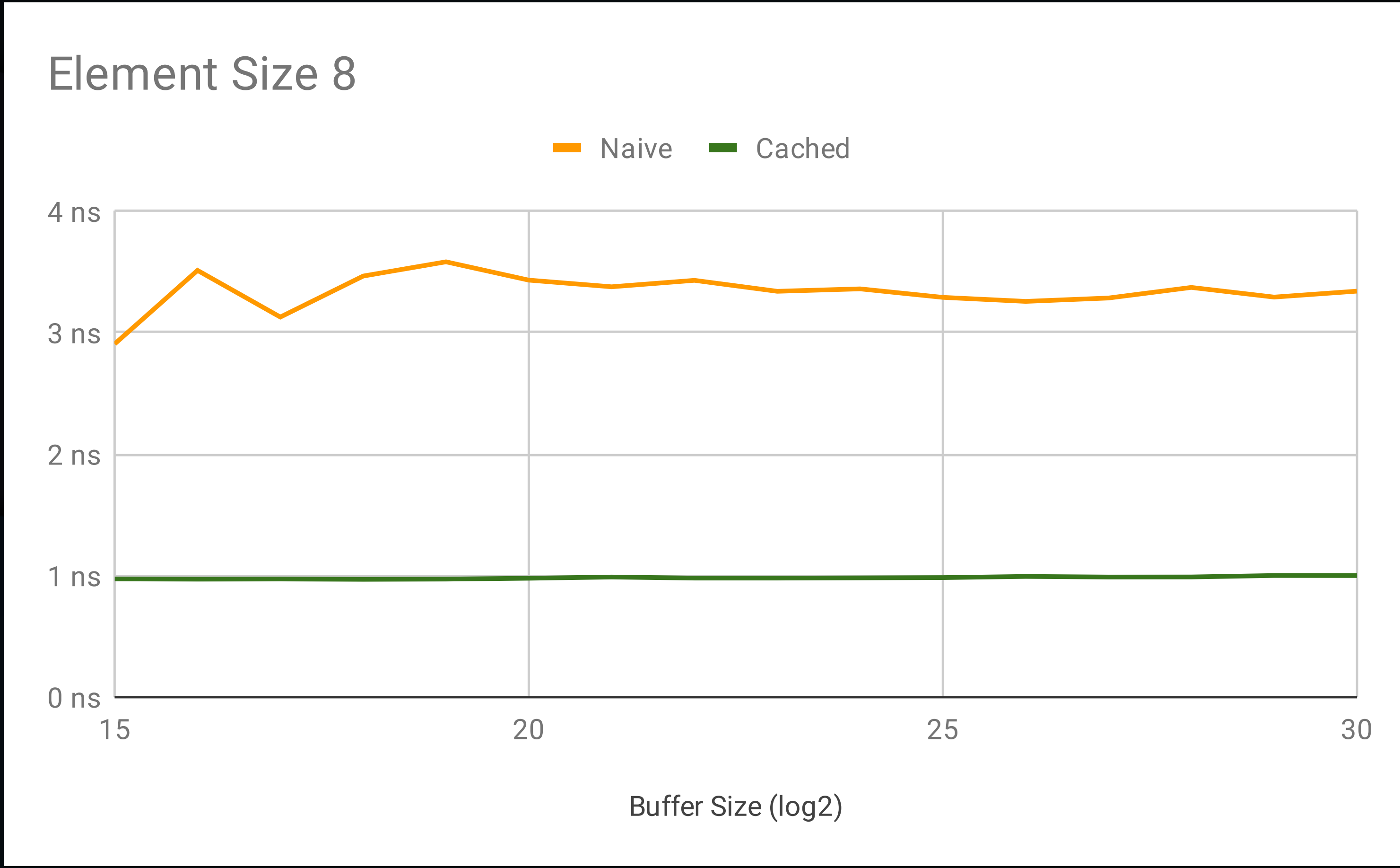
Caching – New Layout



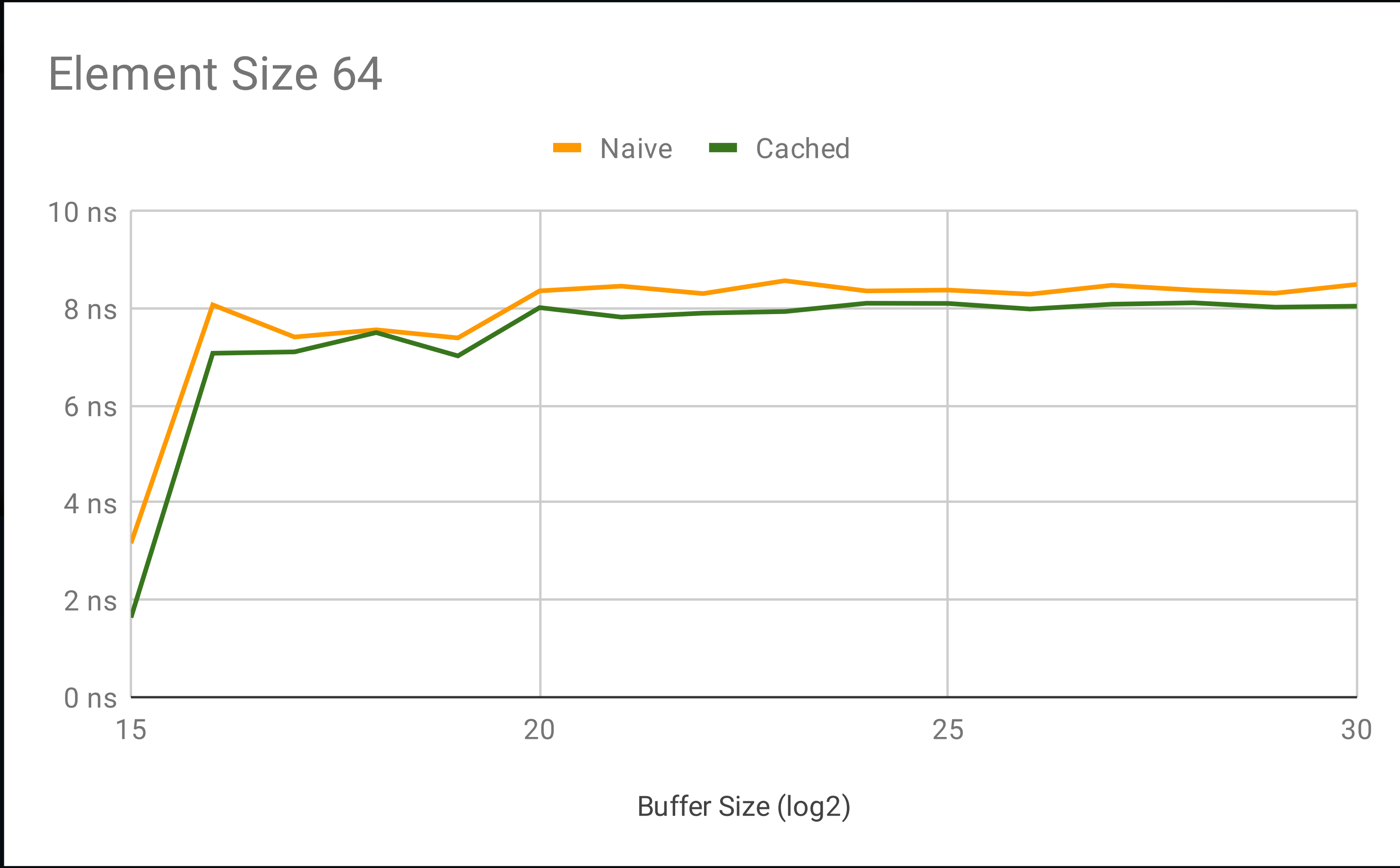
Caching – Produce

```
bool produce(const T& data) {
  auto next = (tail + 1) % SIZE;
  if (next == head_cache)
    if (next == (head_cache = head))
      return false;
  buffer[tail] = data;
  tail = next;
  return true;
}
```

Caching – Benchmark



Caching – Benchmark



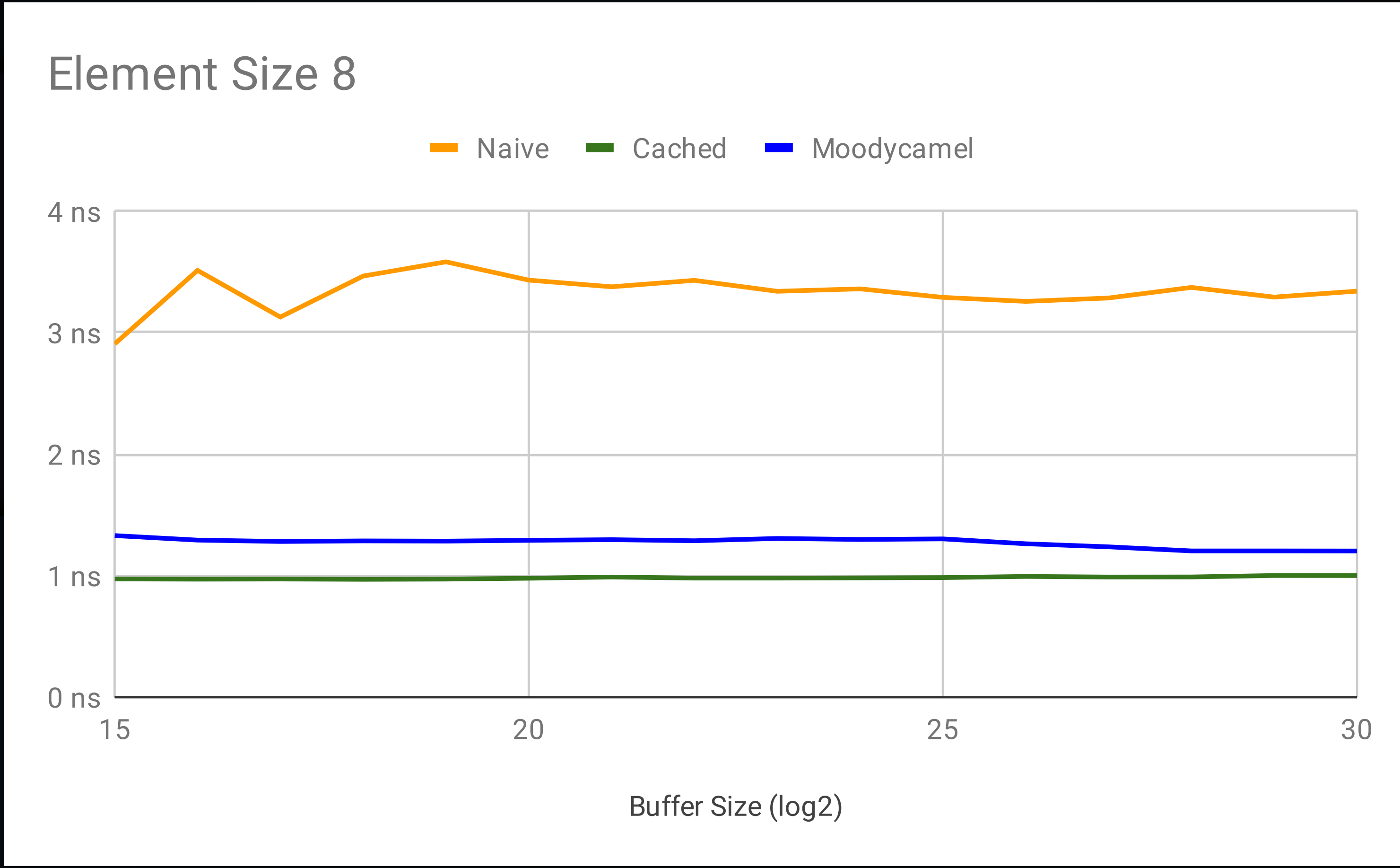
Caching

- First documented in a paper in 2009
- Improves average case, worsens worst case
- Self-balancing
  - If empty, consumer falls into worst case more often
  - If full, producer falls into worst case more often

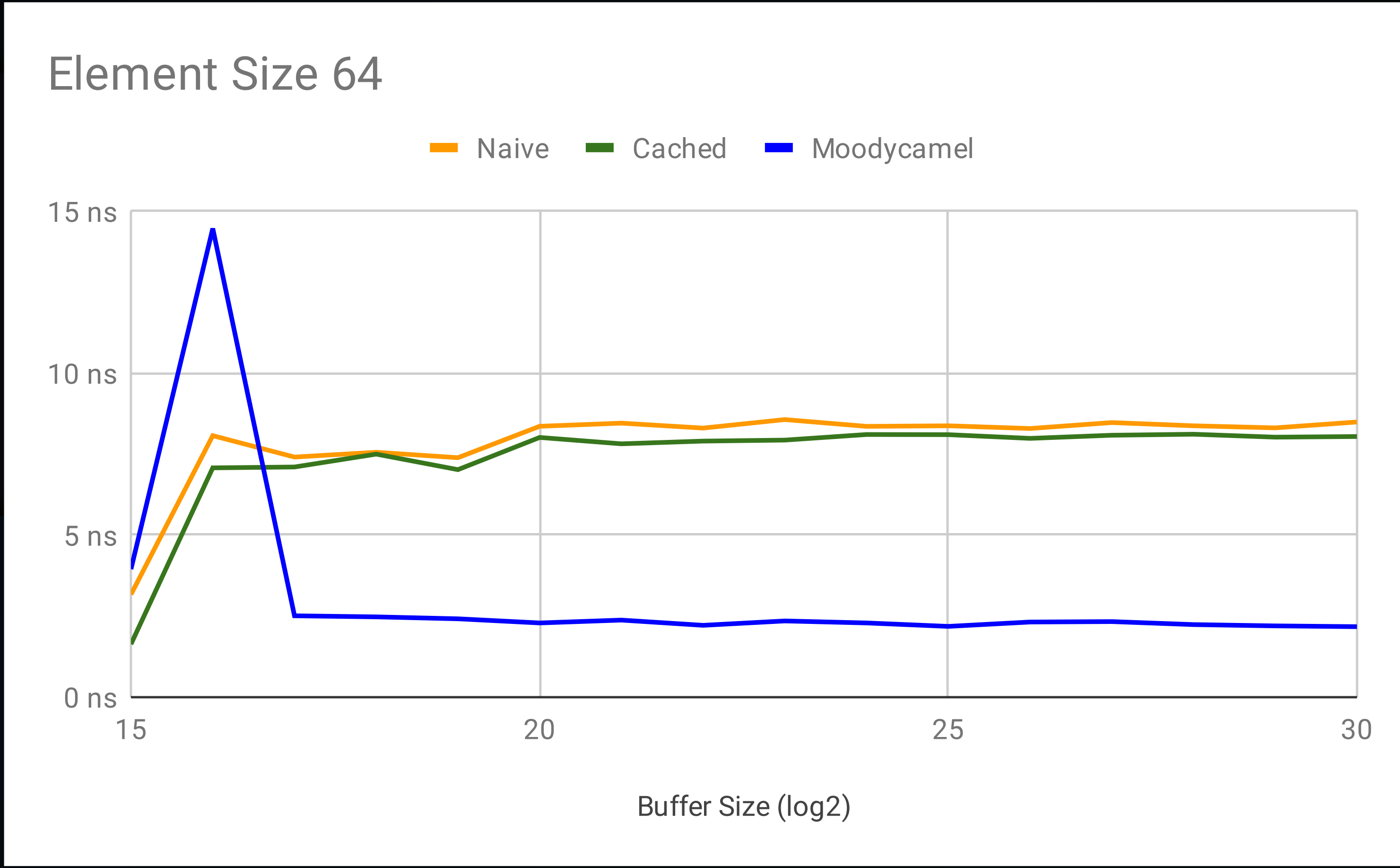
Moodycamel

- Unbounded SP/SC Queue
- Usable as both bounded and unbounded
- Performance comparable to or better than existing implementations

Moodycamel – Benchmarks



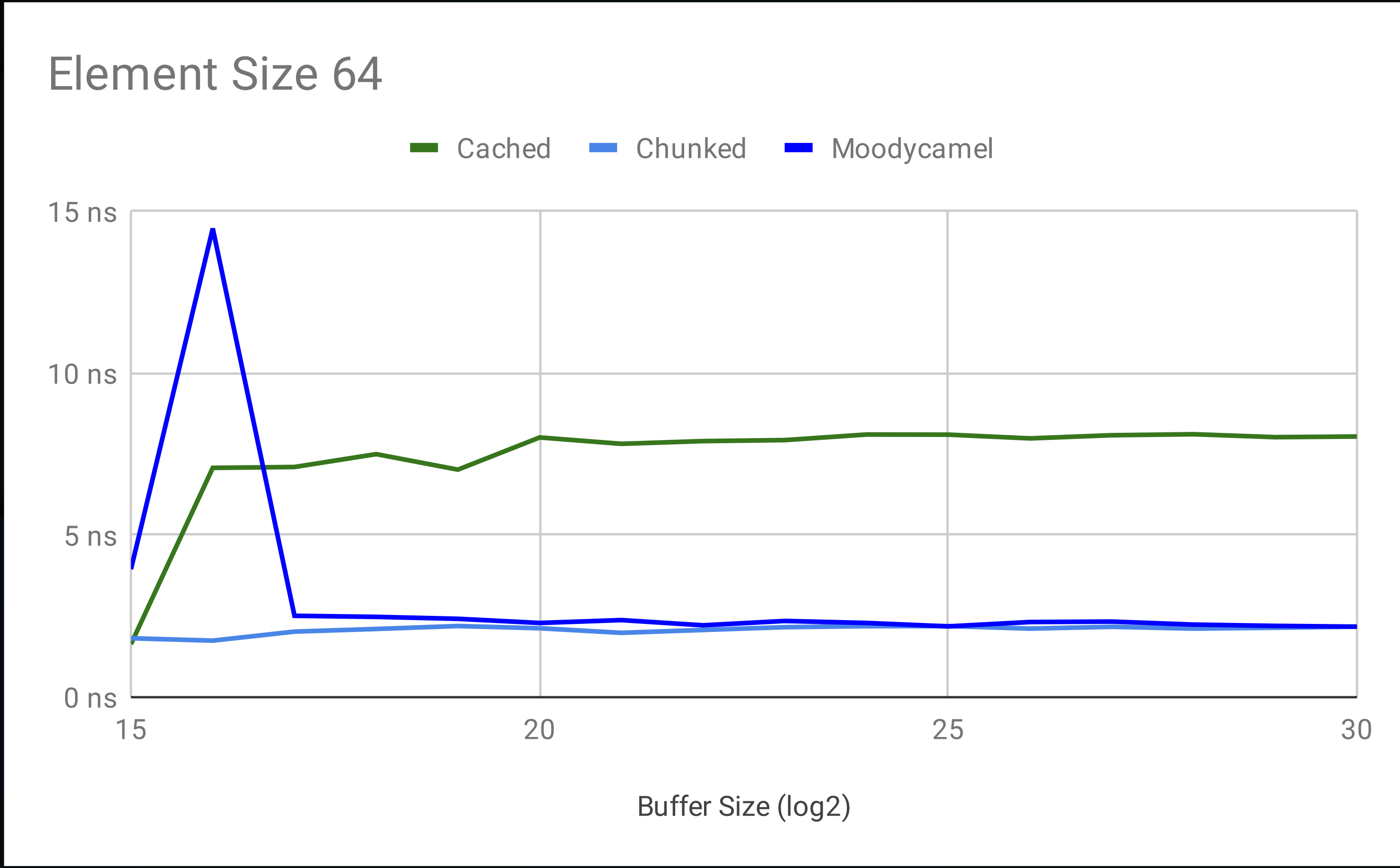
Moodycamel – Benchmarks



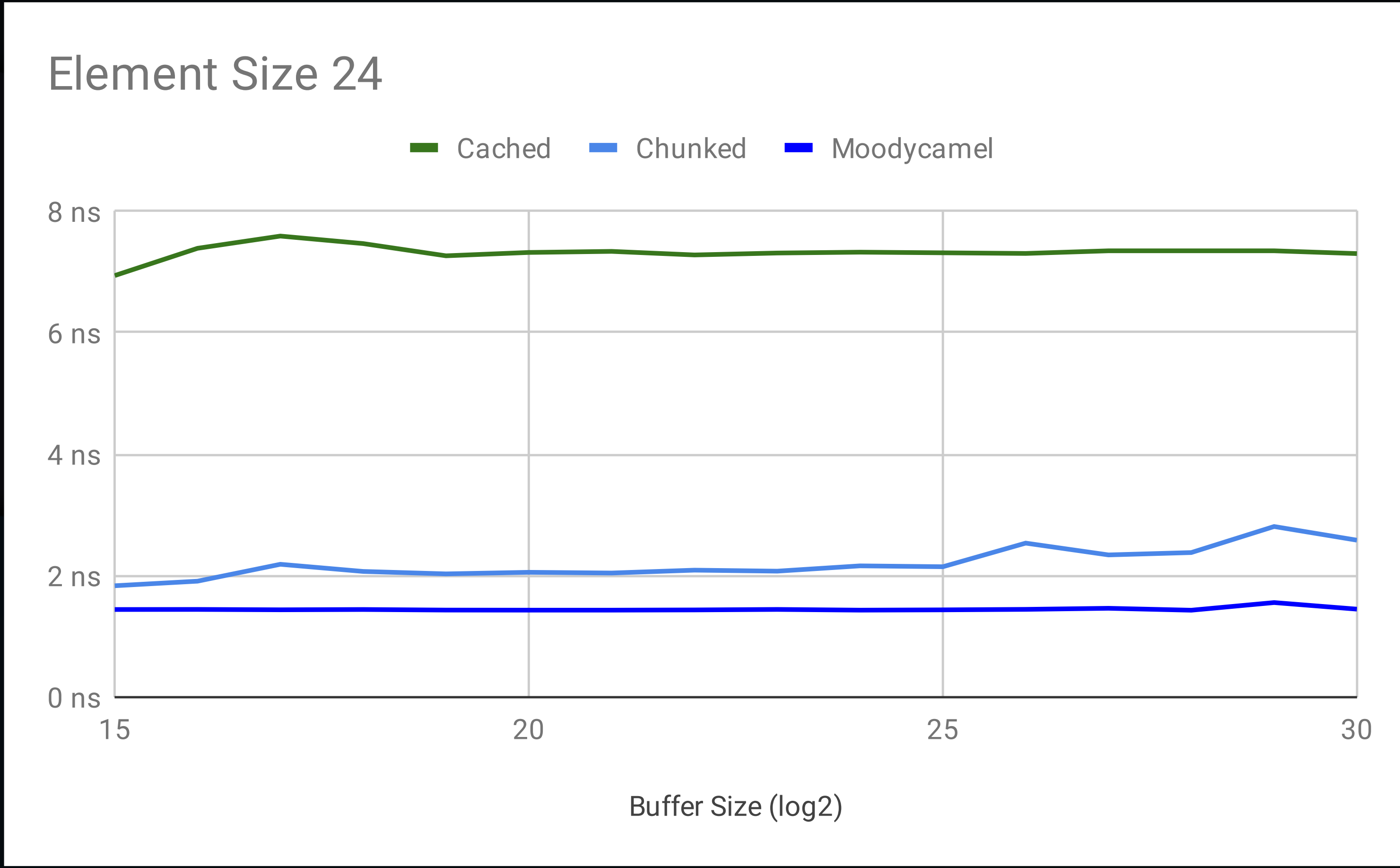
Why is it faster?

- Splitting Queue into Chunks
  - Each Chunk has its own Produce/Consume position
  - Each Chunk has a pointer to the next Chunk
- More code, but with right chunk size it can stay as fast as small queues
  - Right size is <= L1D size

Chunked – Benchmark



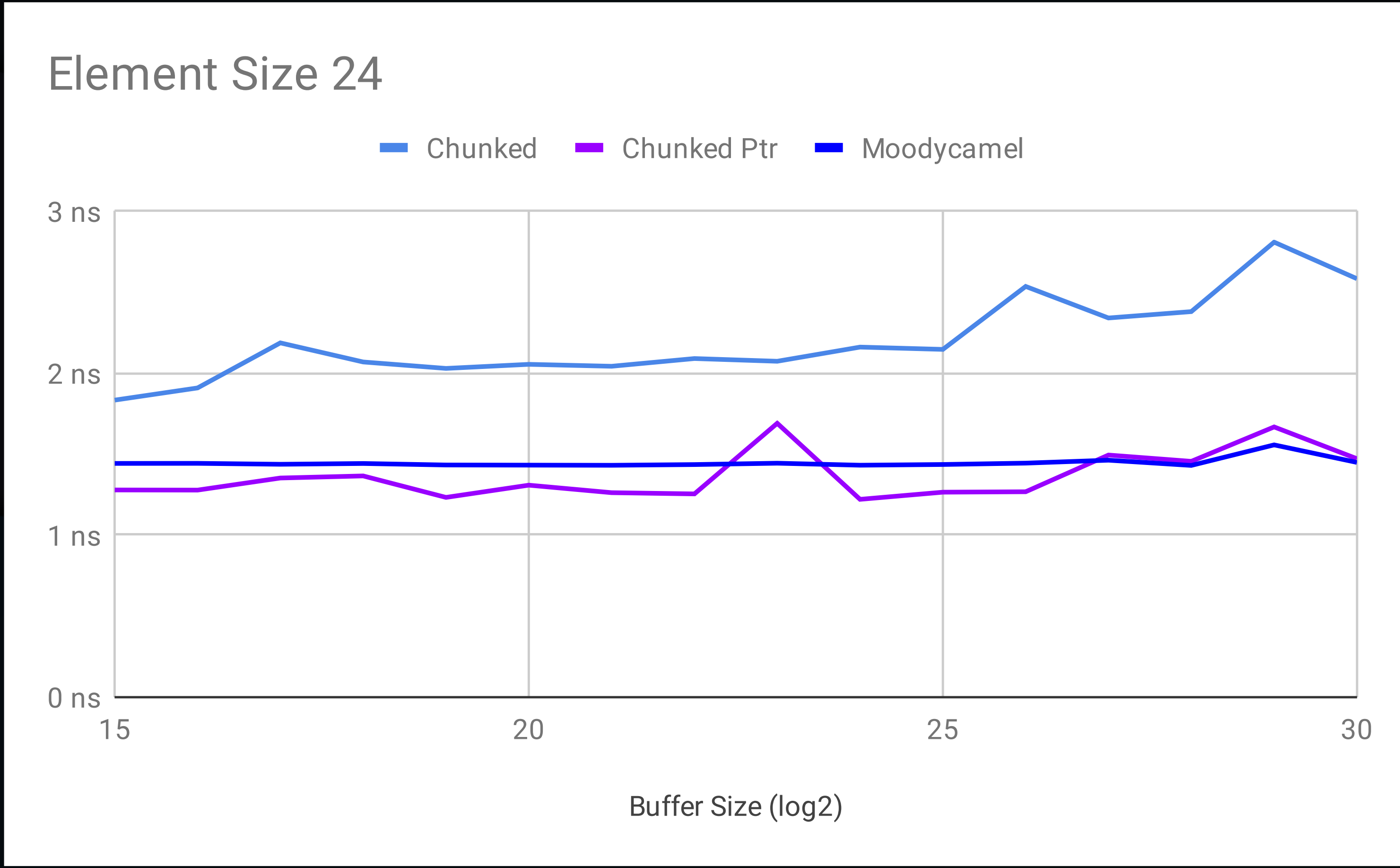
Chunked – Benchmark



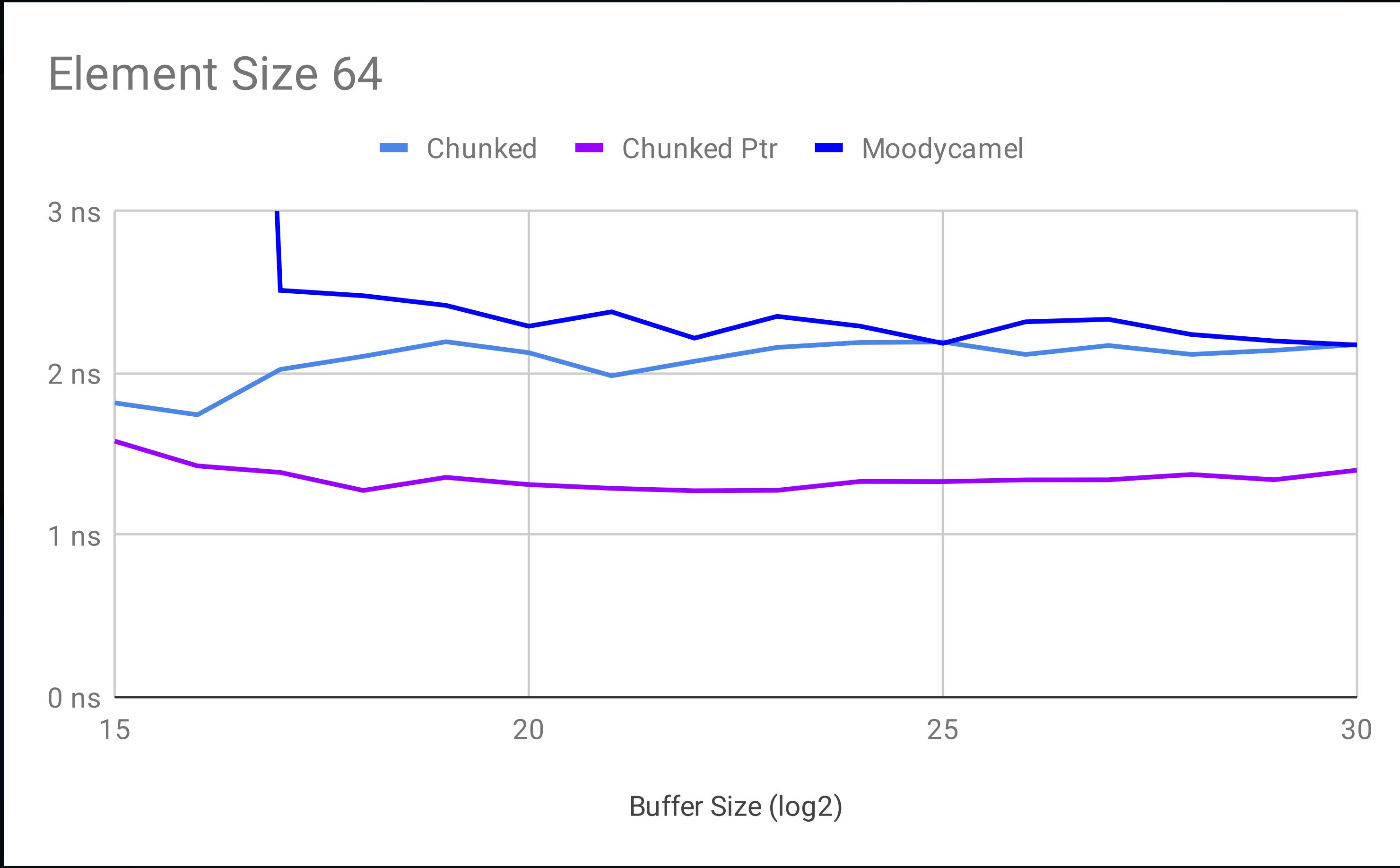
Using Pointers instead of Indices

- Avoid multiplying index with element size
- Forces ability to use arbitrary queue sizes

Pointers – Benchmark



Pointers – Benchmark



Conclusions

- Cache Produce/Consume position for better performance
- Keep Queue small enough to fit into L1D (32KiB on recent x64 processors)
- If you need a bigger Queue, implement Queue of (small) Queues

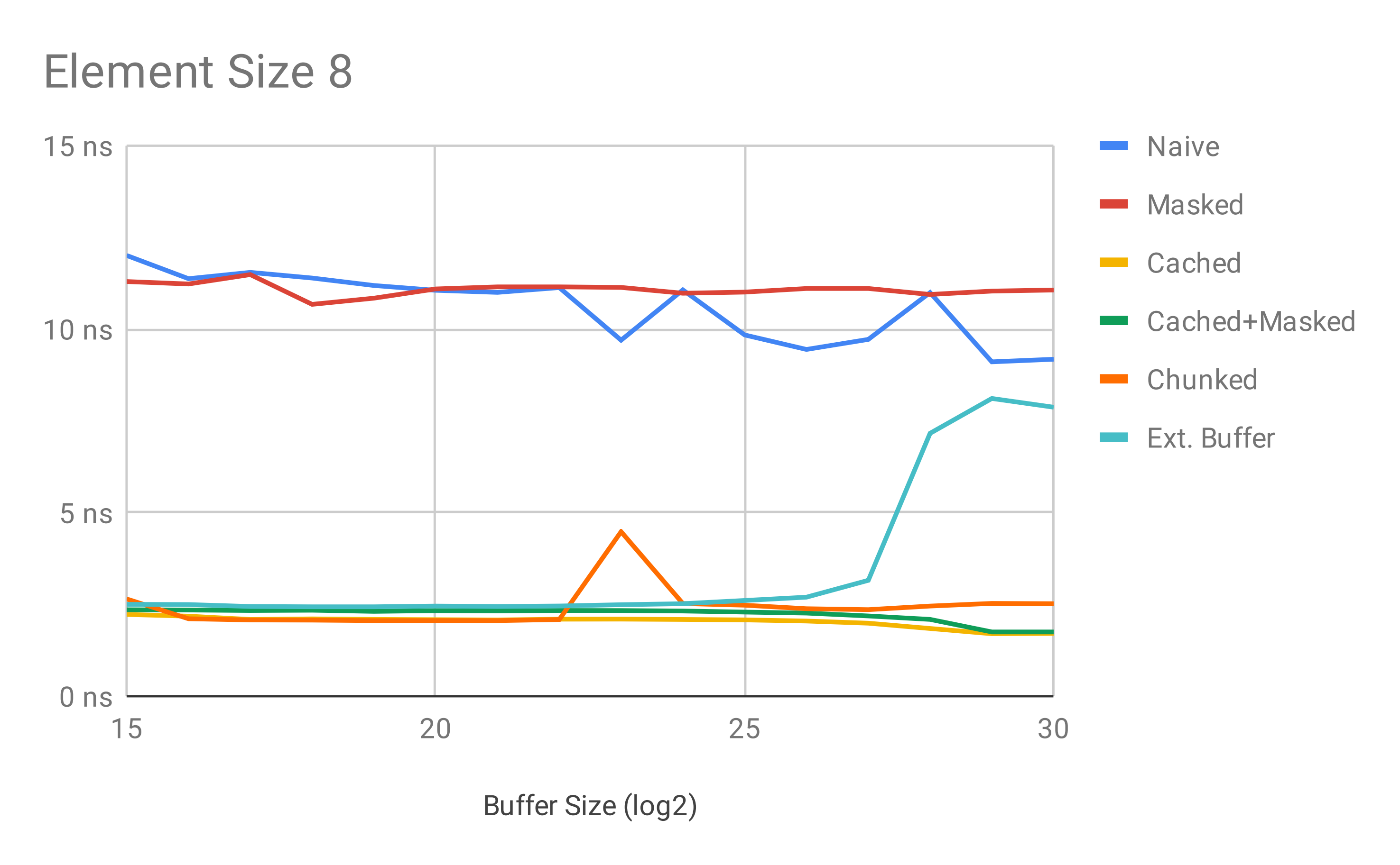
Thank You!

Questions?

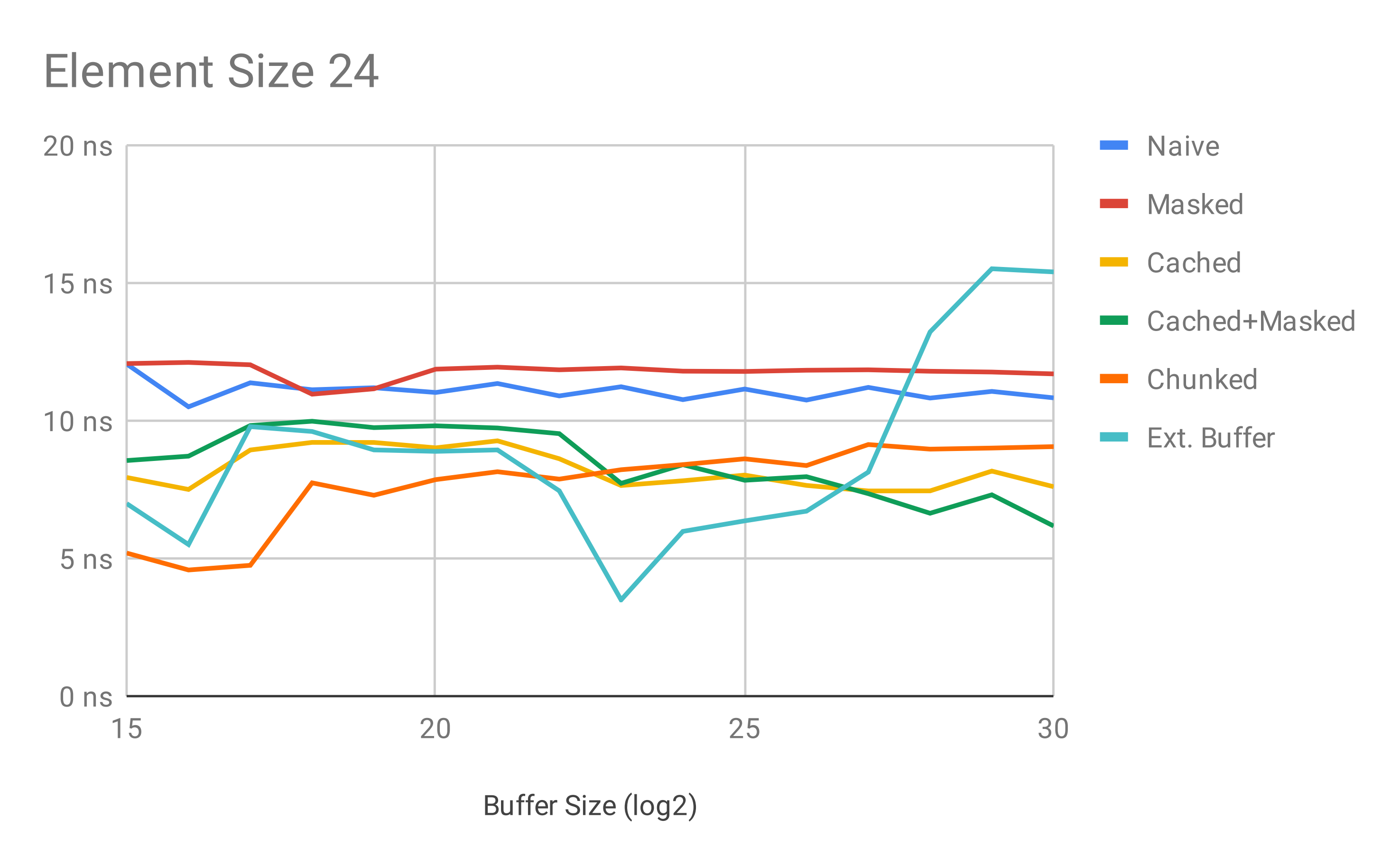
Benchmarks – Ring Buffer

- 8 Bytes overhead per Element
- Elements uninitialized
- Element sizes take overhead into account

Benchmarks

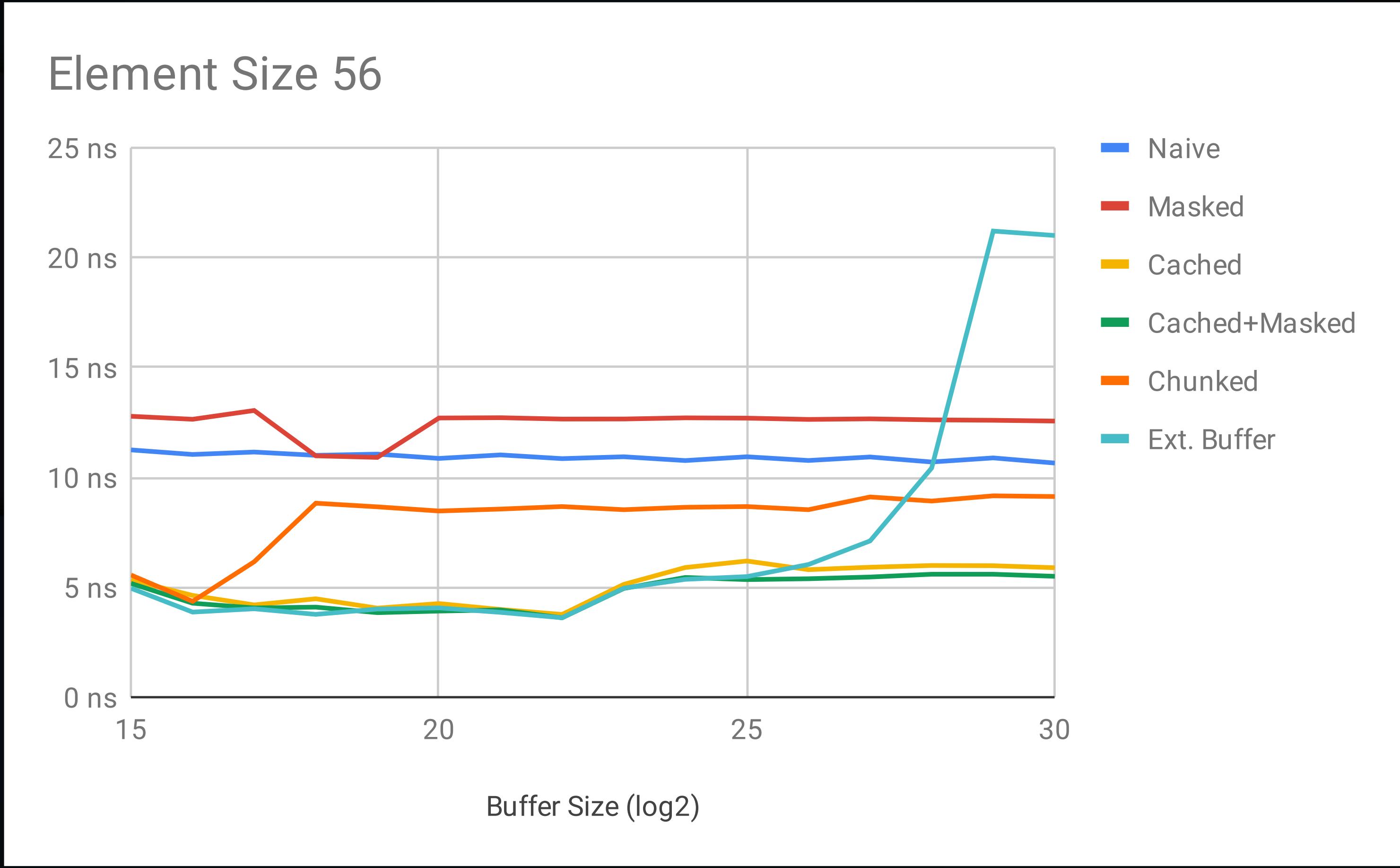

Element Size 8
15 ns
10 ns
5 ns
0 ns
15
20
25
30
Buffer Size (log2)
Naive
Masked
Cached
Cached+Masked
Chunked
Ext. Buffer

Benchmarks



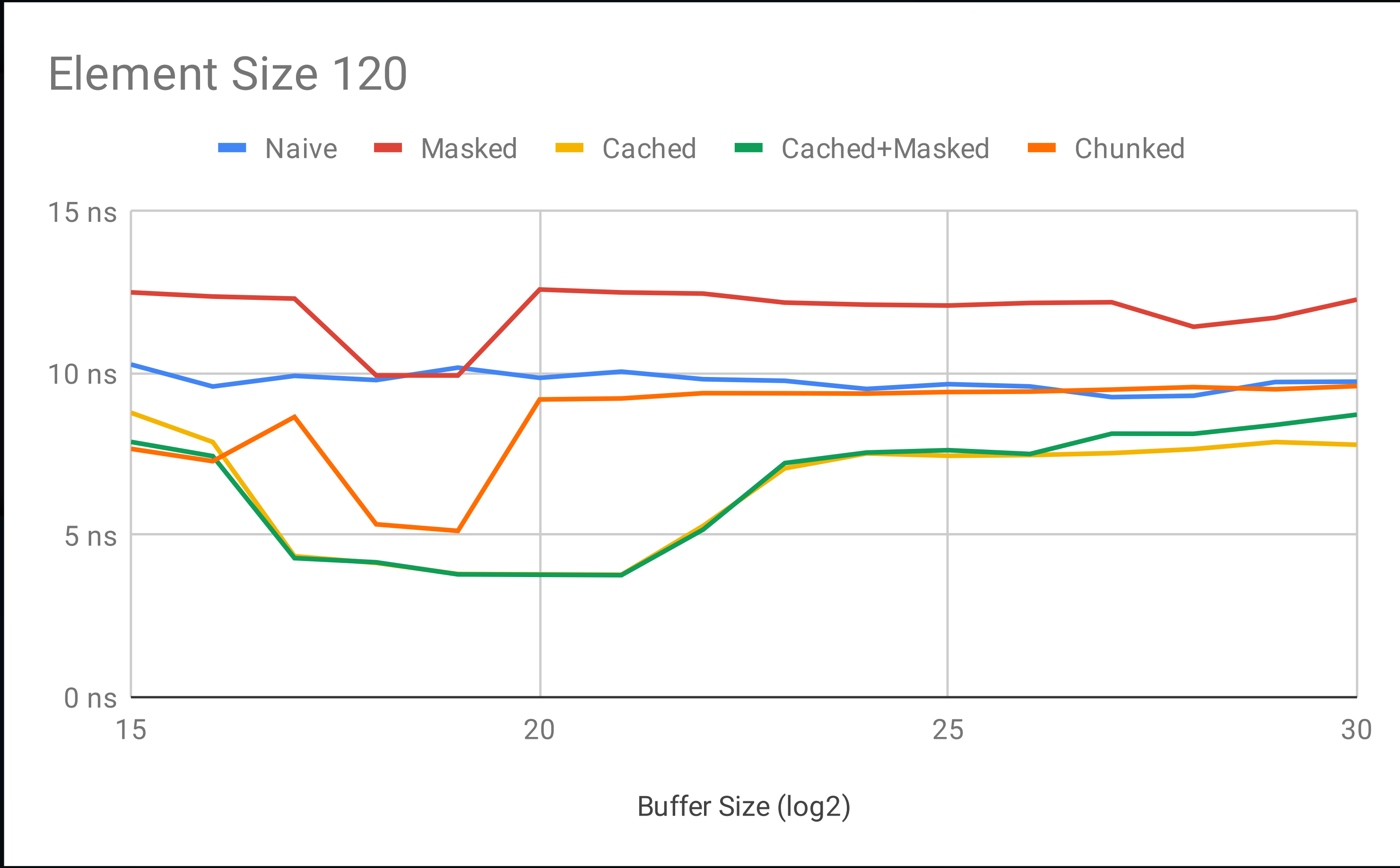
Element Size 24
20 ns
15 ns
10 ns
5 ns
0 ns
15
20
25
30
Buffer Size (log2)
Naive
Masked
Cached
Cached+Masked
Chunked
Ext. Buffer

Benchmarks



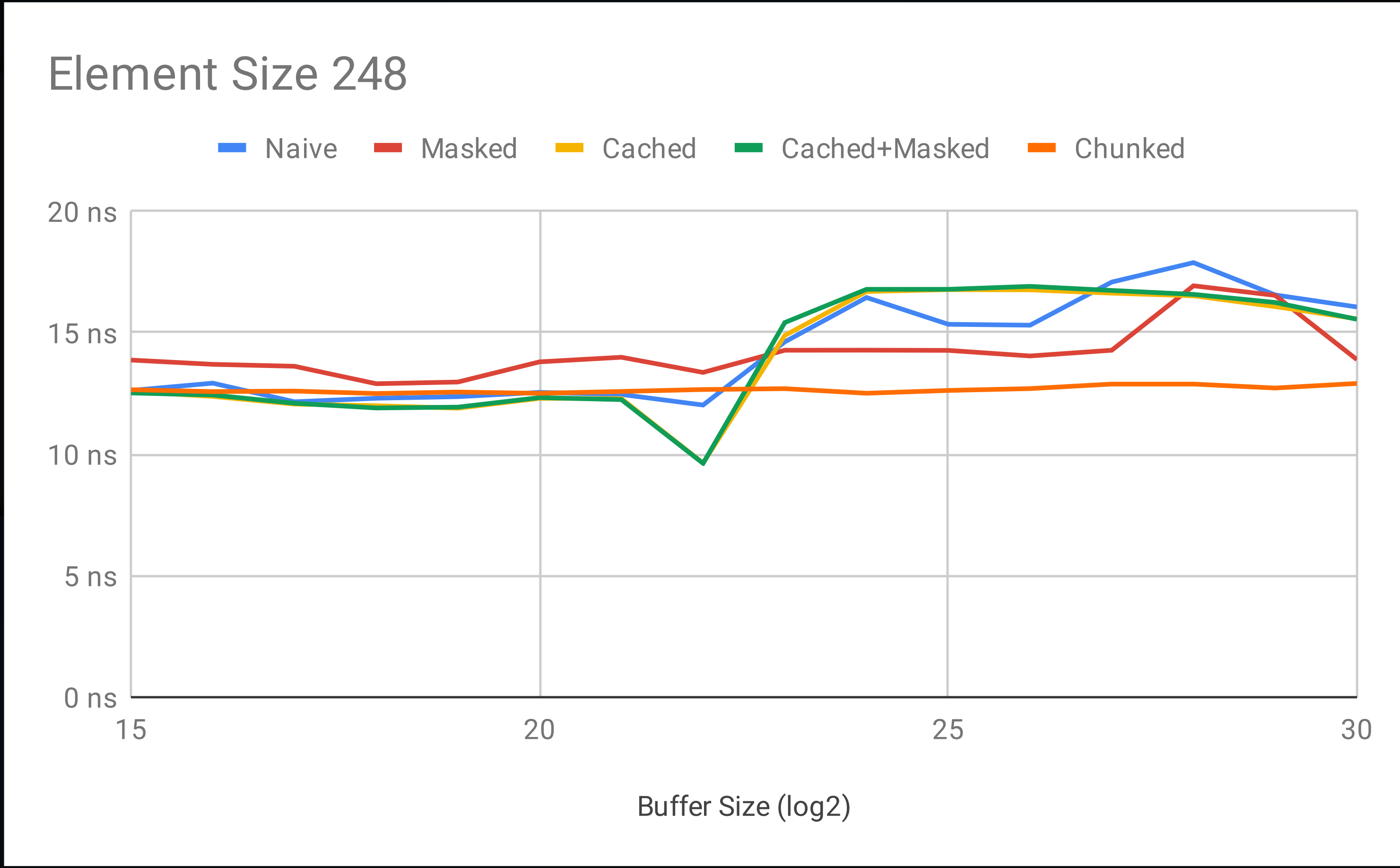
Element Size 56
25 ns
20 ns
15 ns
10 ns
5 ns
0 ns
15
20
25
30
Buffer Size (log2)
Naive
Masked
Cached
Cached+Masked
Chunked
Ext. Buffer

Benchmarks



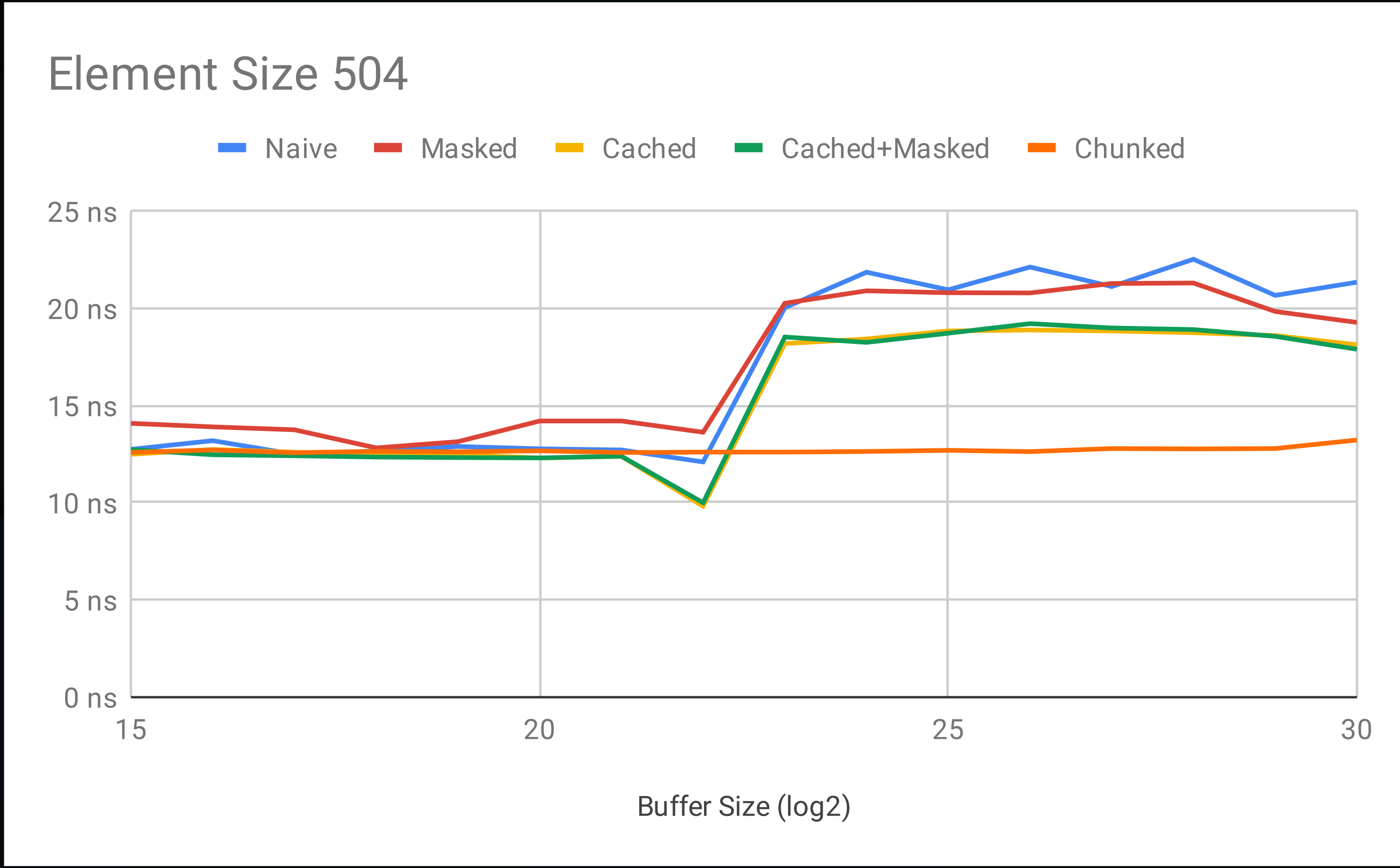
Element Size 120
Naive
Masked
Cached
Cached+Masked
Chunked
15 ns
10 ns
5 ns
0 ns
15
20
25
30
Buffer Size (log2)

Benchmarks



Element Size 248
Naive
Masked
Cached
Cached+Masked
Chunked
20 ns
15 ns
10 ns
5 ns
0 ns
15
20
25
30
Buffer Size (log2)

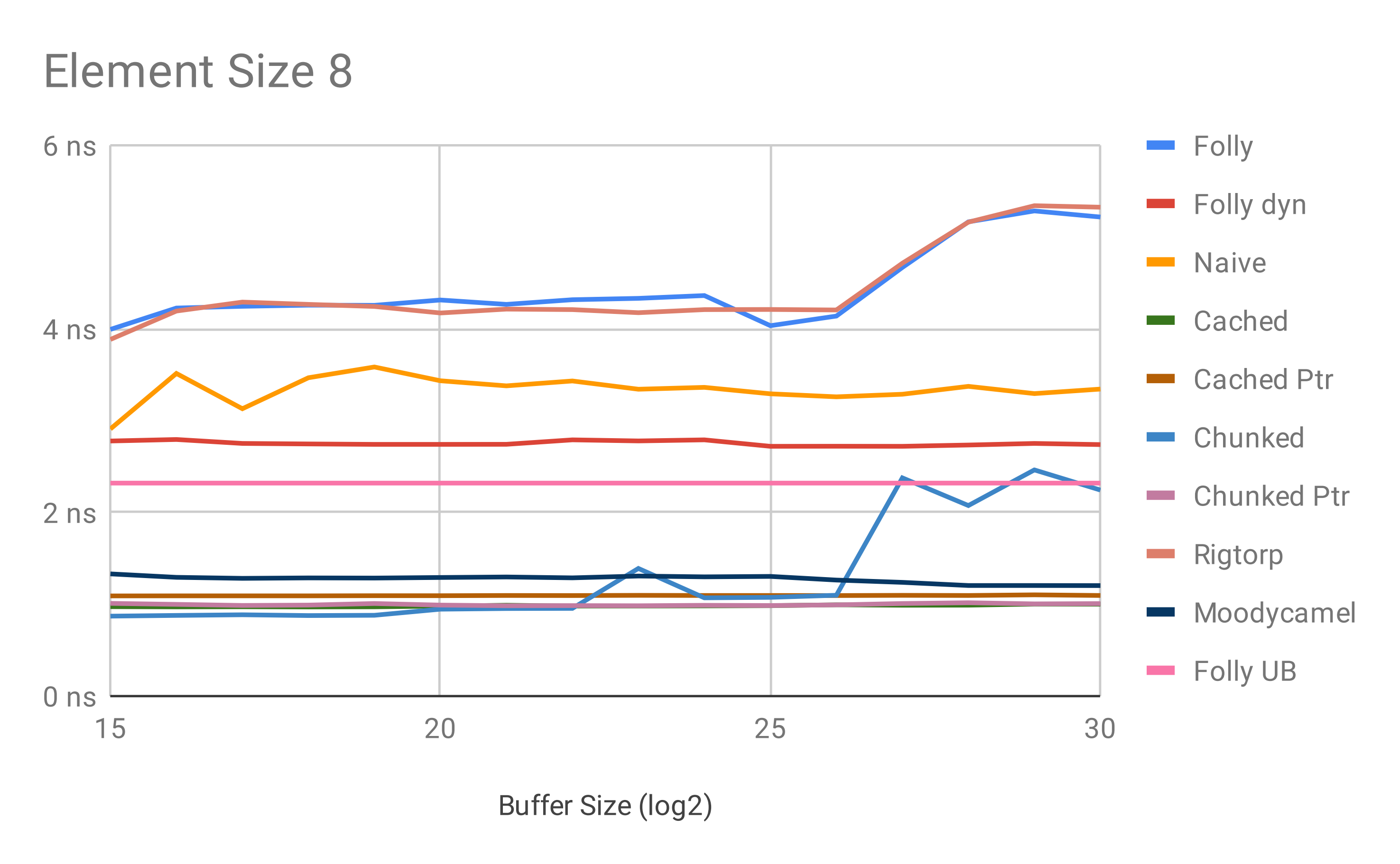
Benchmarks


Element Size 504
Naive
Masked
Cached
Cached+Masked
Chunked
25 ns
20 ns
15 ns
10 ns
5 ns
0 ns
15
20
25
30
Buffer Size (log2)

Benchmarks – Queue

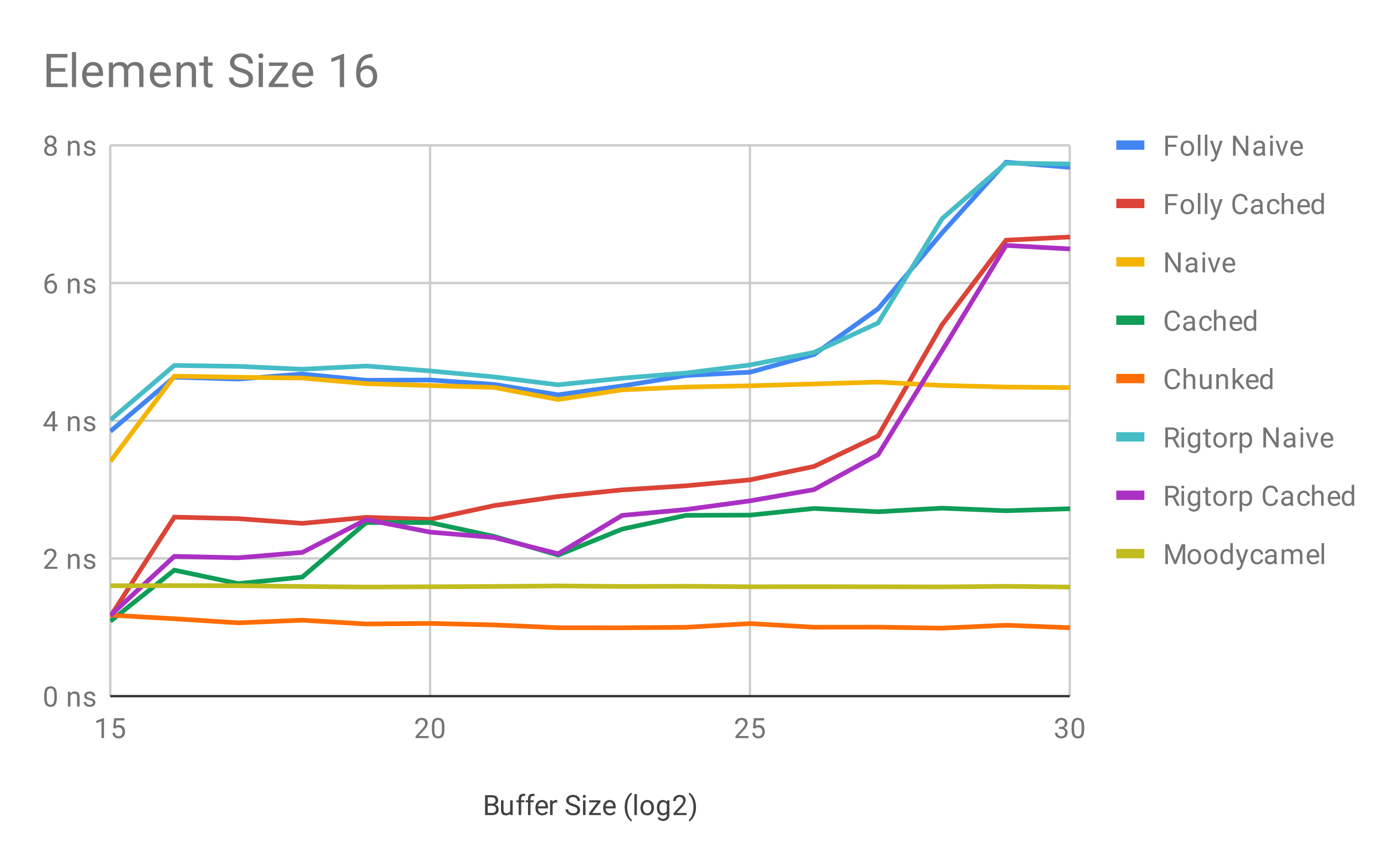
- No overhead per element
- Elements are initialized

Benchmarks



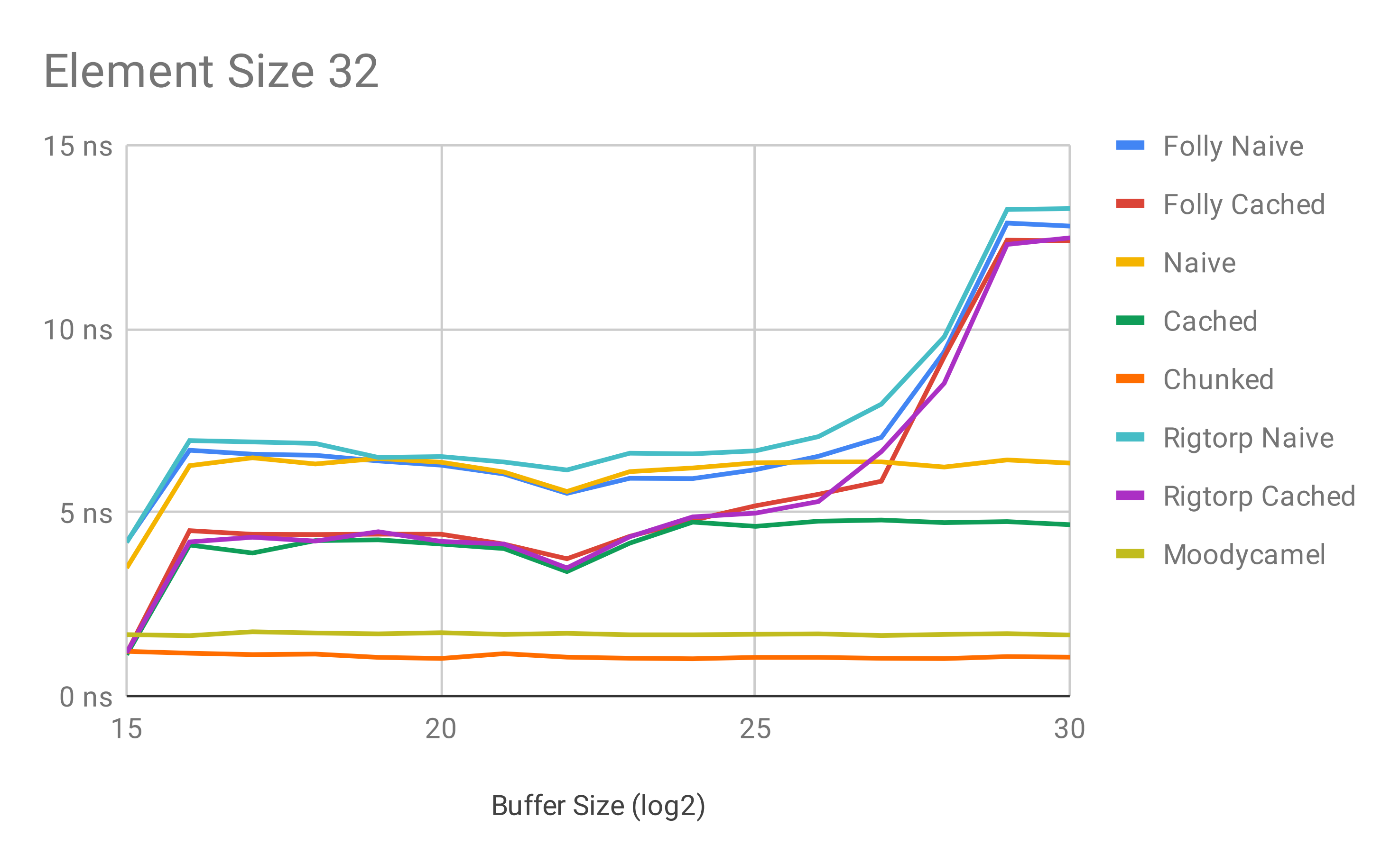
Element Size 8
6 ns
4 ns
2 ns
0 ns
15
20
25
30
Buffer Size (log2)
Folly
Folly dyn
Naive
Cached
Cached Ptr
Chunked
Chunked Ptr
Rigtorp
Moodycamel
Folly UB

Benchmarks



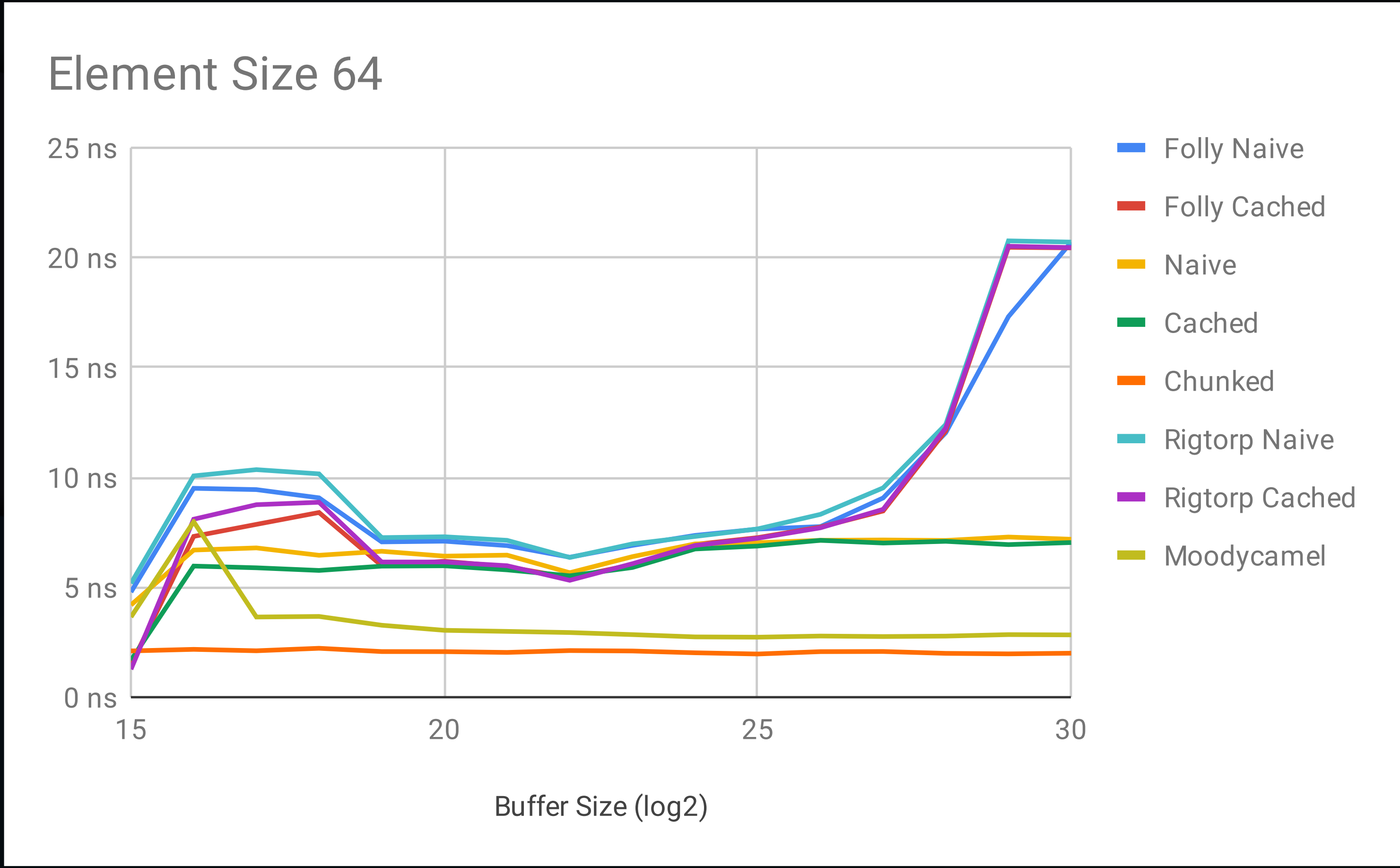
Element Size 16
8 ns
6 ns
4 ns
2 ns
0 ns
15
20
25
30
Buffer Size (log2)
Folly Naive
Folly Cached
Naive
Cached
Chunked
Rigtorp Naive
Rigtorp Cached
Moodycamel

Benchmarks



Element Size 32
15 ns
10 ns
5 ns
0 ns
15
20
25
30
Buffer Size (log2)
Folly Naive
Folly Cached
Naive
Cached
Chunked
Rigtorp Naive
Rigtorp Cached
Moodycamel

Benchmarks



Element Size 64
25 ns
20 ns
15 ns
10 ns
5 ns
0 ns
15
20
25
30
Buffer Size (log2)
Folly Naive
Folly Cached
Naive
Cached
Chunked
Rigtorp Naive
Rigtorp Cached
Moodycamel

Links

Ring Buffer Benchmark Results: https://docs.google.com/spreadsheets/d/1KOw_3-6XaX1No4j5QUmCYYbUTKDtRosy4aYOJYyOikU/edit#gid=128804321

Queue Benchmark Results: https://docs.google.com/spreadsheets/d/1Bb_ClWBmr3XqJHGOK_mJUmyDNyzDOs0-dy7zaeFHJNA/edit#gid=877059096

GitHub Repository: https://github.com/Deaod/RingBufferBenchmark

MCRingBuffer Paper: http://www.cse.cuhk.edu.hk/~pclee/www/pubs/ancs09poster.pdf

As an Aside

- Overclocking Memory is a Bad Idea
- Overclocking Dense Memory Doubly so
- Therefore … More Voltage
- Turns out I wrote a Memory Test