Ring Buffers As Fast As Possible

Content

- Motivation
- Queue Basics
- Possible Trade-Offs
- Going Fast
- Benchmarks

Terminology

- Single Producer/Consumer Bounded FIFO 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
  - No syscalls
    - No memory allocation
    - No mutexes
  - Lock-free algorithms
    - Wait-free would be best

Motivation

- Separate thread for logging needed
- Queue needed to move data across threads
- Ring Buffer is a straight-forward Queue

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
- Consume Position

Queue Basics – Produce

```
bool produce(…) {
  auto next = (m_produce_pos + 1) % SIZE;
  if (next == m_consume_pos) return 0;
  new(m_buffer + m_produce_pos) T(…);
  m_produce_pos = next;
  return 1;
}
```

Queue Basics – Consume

```
bool consume(Callable f) {
 if (m_consume_pos == m_produce_pos)
  return 0;
 f(m_buffer[m_consume_pos]);
 m_consume_pos = (m_consume_pos+1) % SIZE;
 return 1;
}
```

Possible Trade-Offs

- Multiple vs. Single Producers/Consumers
  - Equivalent to lock-free vs. wait-free
    - Maybe not technically correct
  - Affects complexity of implementation

Possible Trade-Offs

- Fixed vs. variable size of elements
  - Changes the interface
  - Variable element size requires overhead in buffer

Ring Buffer Interface

- Basic idea: No copies

  Not this:

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

Ring Buffer Interface – First Attempt

- Basic idea: No copies

  More like this:

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

  Bugs abound

Ring Buffer Interface – Second Attempt

- 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 – Third Attempt

- 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 – Final Attempt

- Basic idea: No copies

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

  Least bad option, requires use of placement-new

Queue Interface

- Bad:

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

- Standard:

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

- Mine:

```
template<typename Callable>
bool produce(Callable f);
```

Consume Interface

- Usual:

```
T* front();
void pop();
```

- Boost, mine:

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

Interface – Examples

- Boost::Lockfree
- Folly
- Eric Rigtorp

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

Going Fast

- 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.

- Prevent False Sharing

  Put Produce/Consume position each on their own cache line

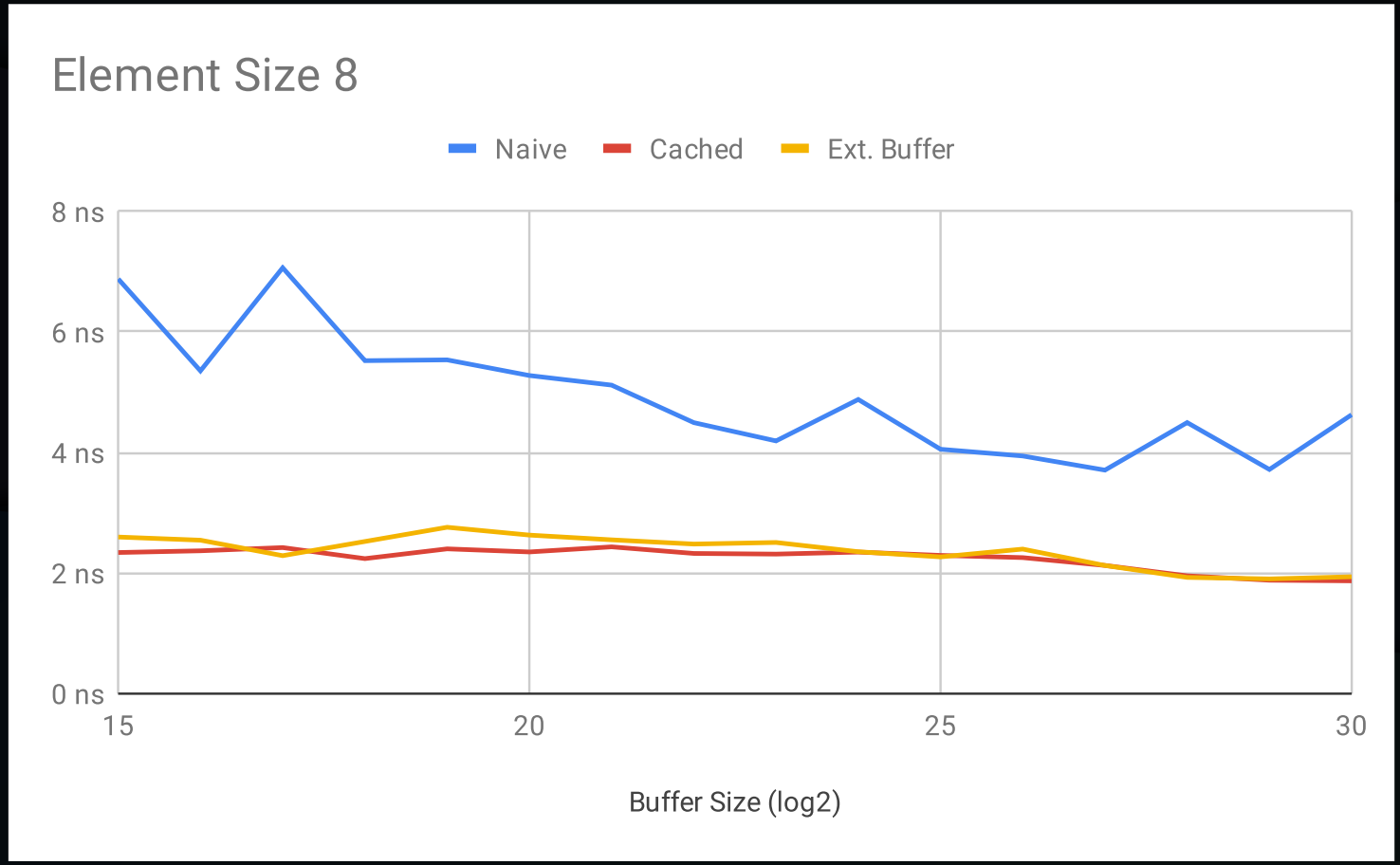
Going Faster

- 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

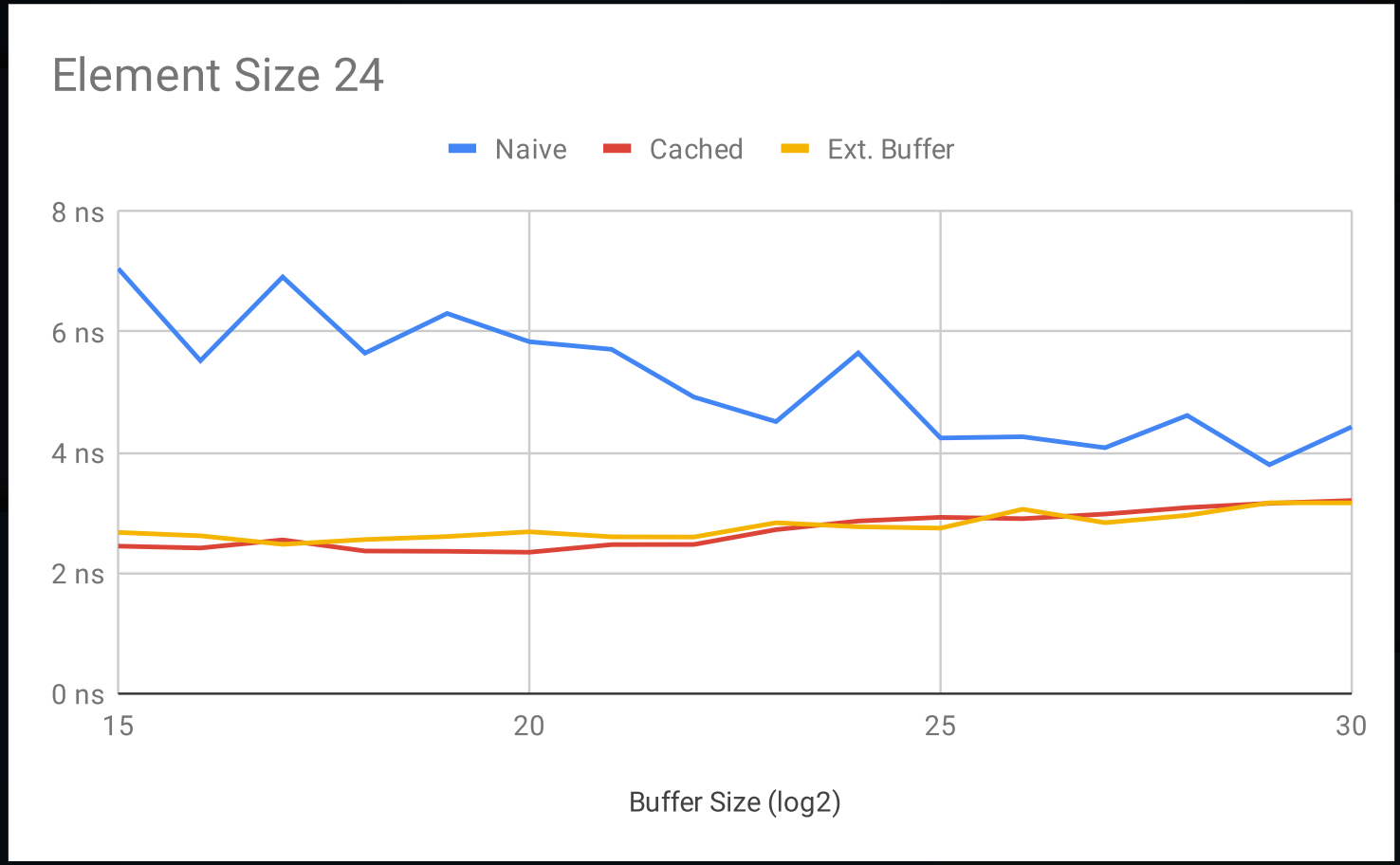
Benchmarks – Ring Buffer

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

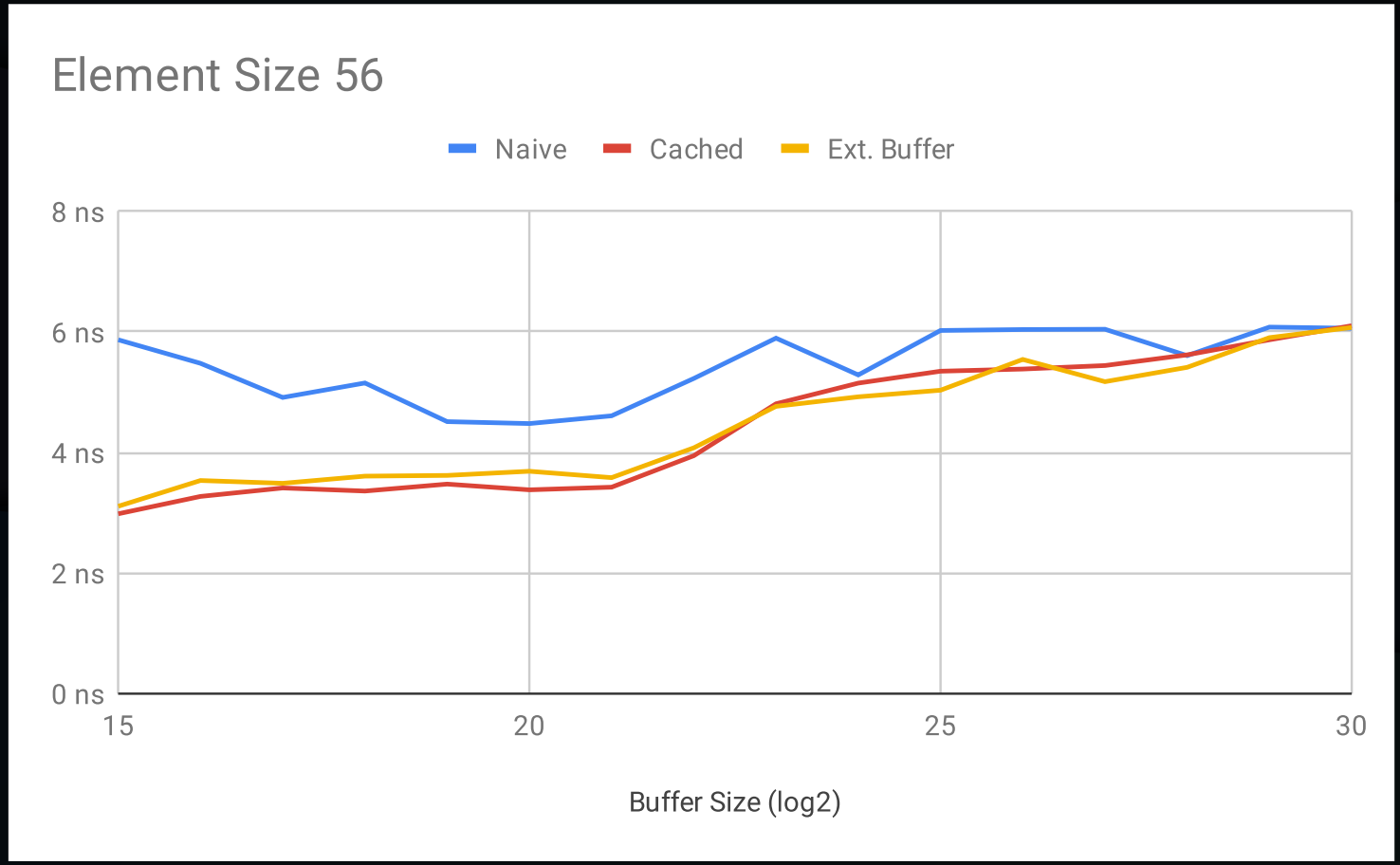
Benchmarks


Element Size 8
Naive
Cached
Ext. Buffer
8 ns
6 ns
4 ns
2 ns
0 ns
15
20
25
30
Buffer Size (log2)

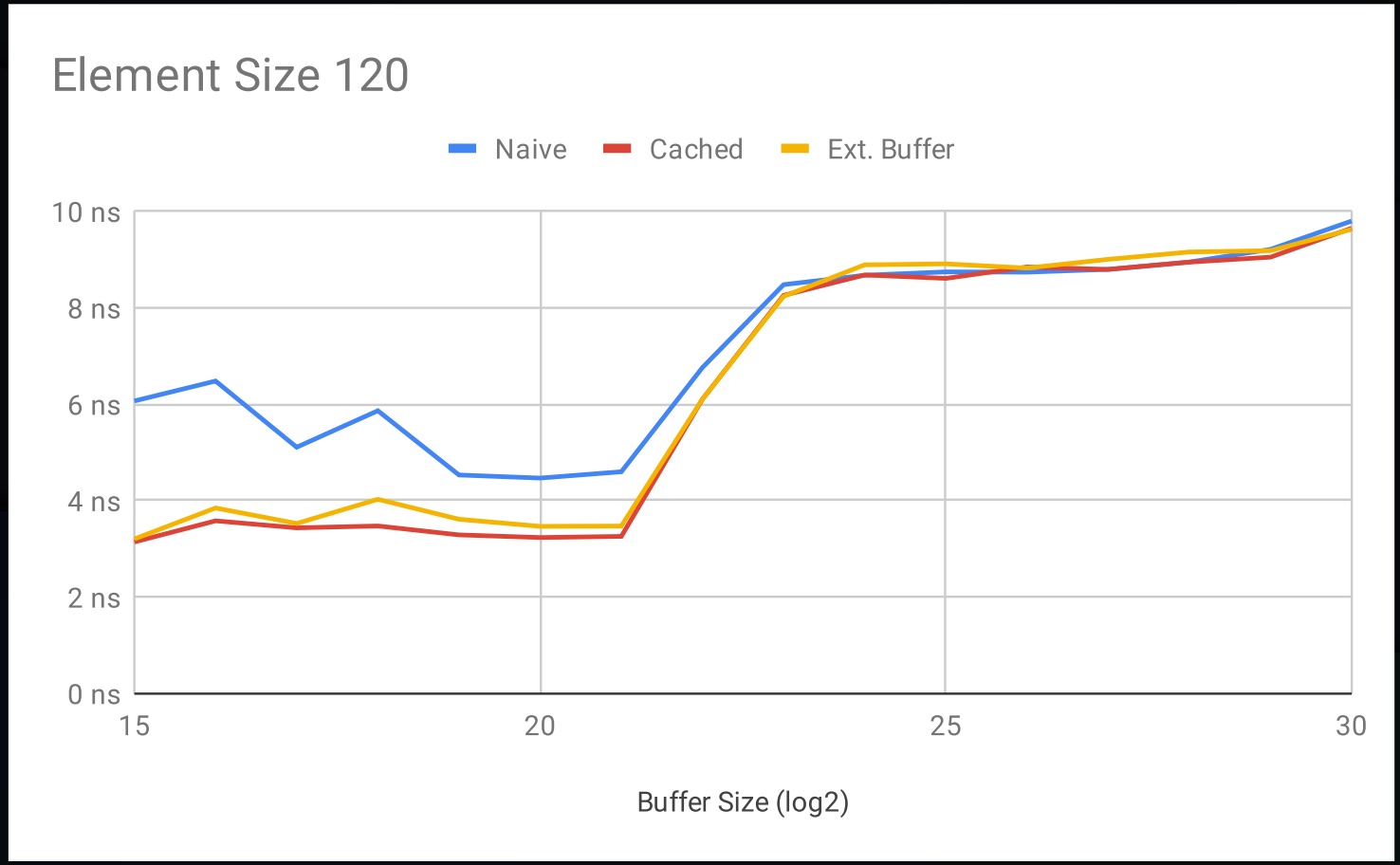
Benchmarks


Element Size 24
Naive
Cached
Ext. Buffer
8 ns
6 ns
4 ns
2 ns
0 ns
15
20
25
30
Buffer Size (log2)

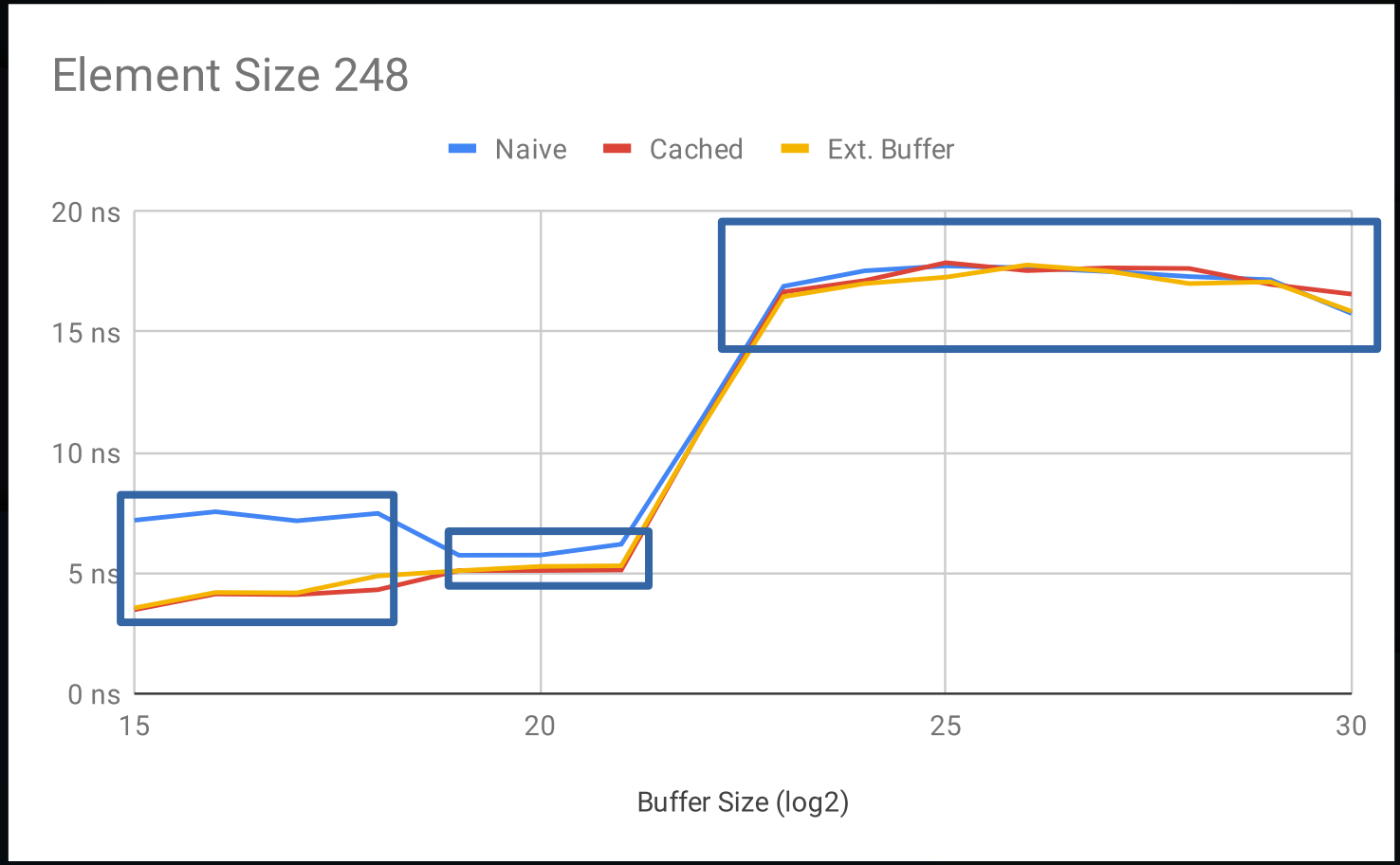
Benchmarks


Element Size 56
Naive
Cached
Ext. Buffer
8 ns
6 ns
4 ns
2 ns
0 ns
15
20
25
30
Buffer Size (log2)

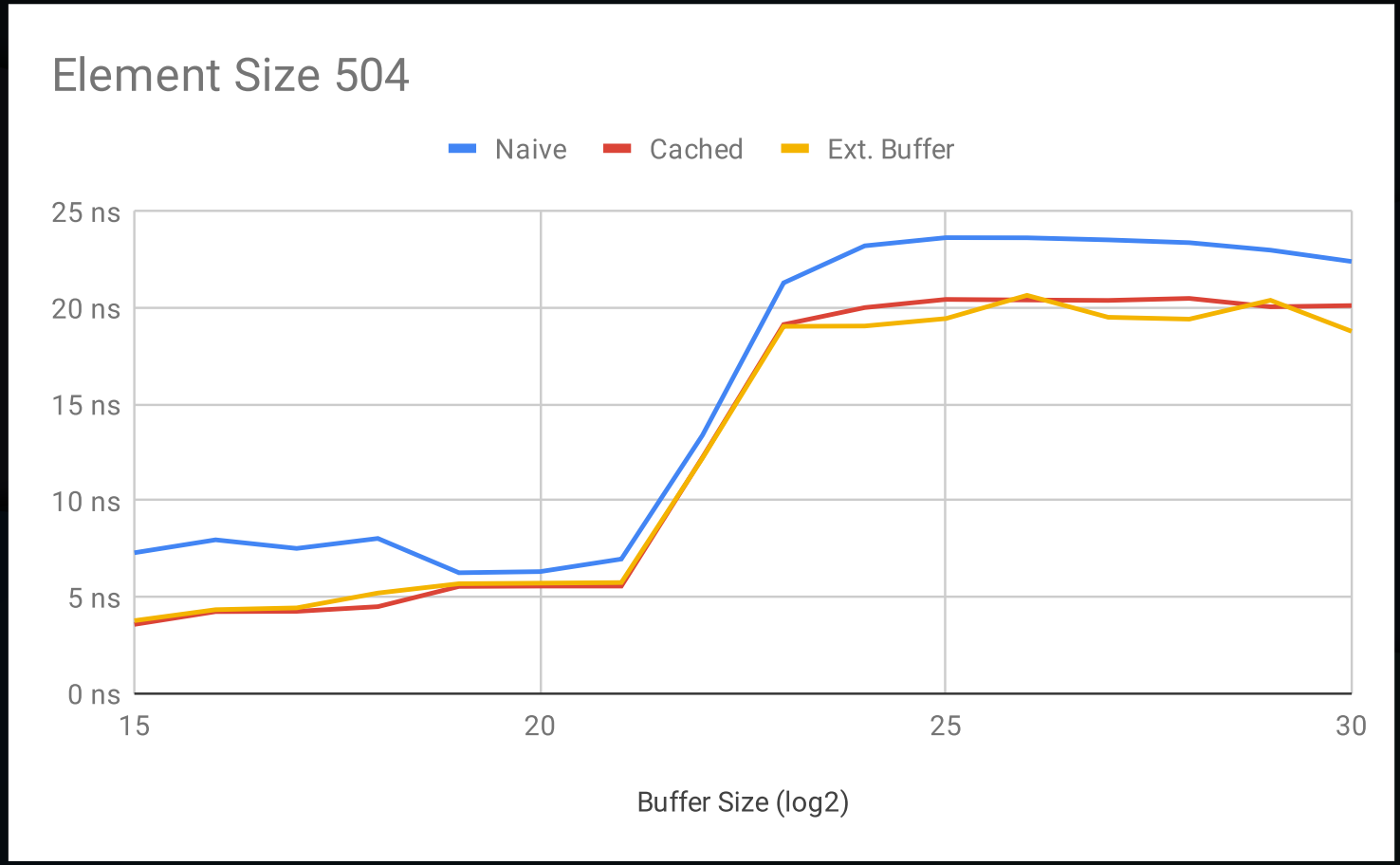
Benchmarks


Element Size 120
Naive
Cached
Ext. Buffer
10 ns
8 ns
6 ns
4 ns
2 ns
0 ns
15
20
25
30
Buffer Size (log2)

Benchmarks


Element Size 248
Naive
Cached
Ext. Buffer
20 ns
15 ns
10 ns
5 ns
0 ns
15
20
25
30
Buffer Size (log2)

Benchmarks


Element Size 504
Naive
Cached
Ext. Buffer
25 ns
20 ns
15 ns
10 ns
5 ns
0 ns
15
20
25
30
Buffer Size (log2)

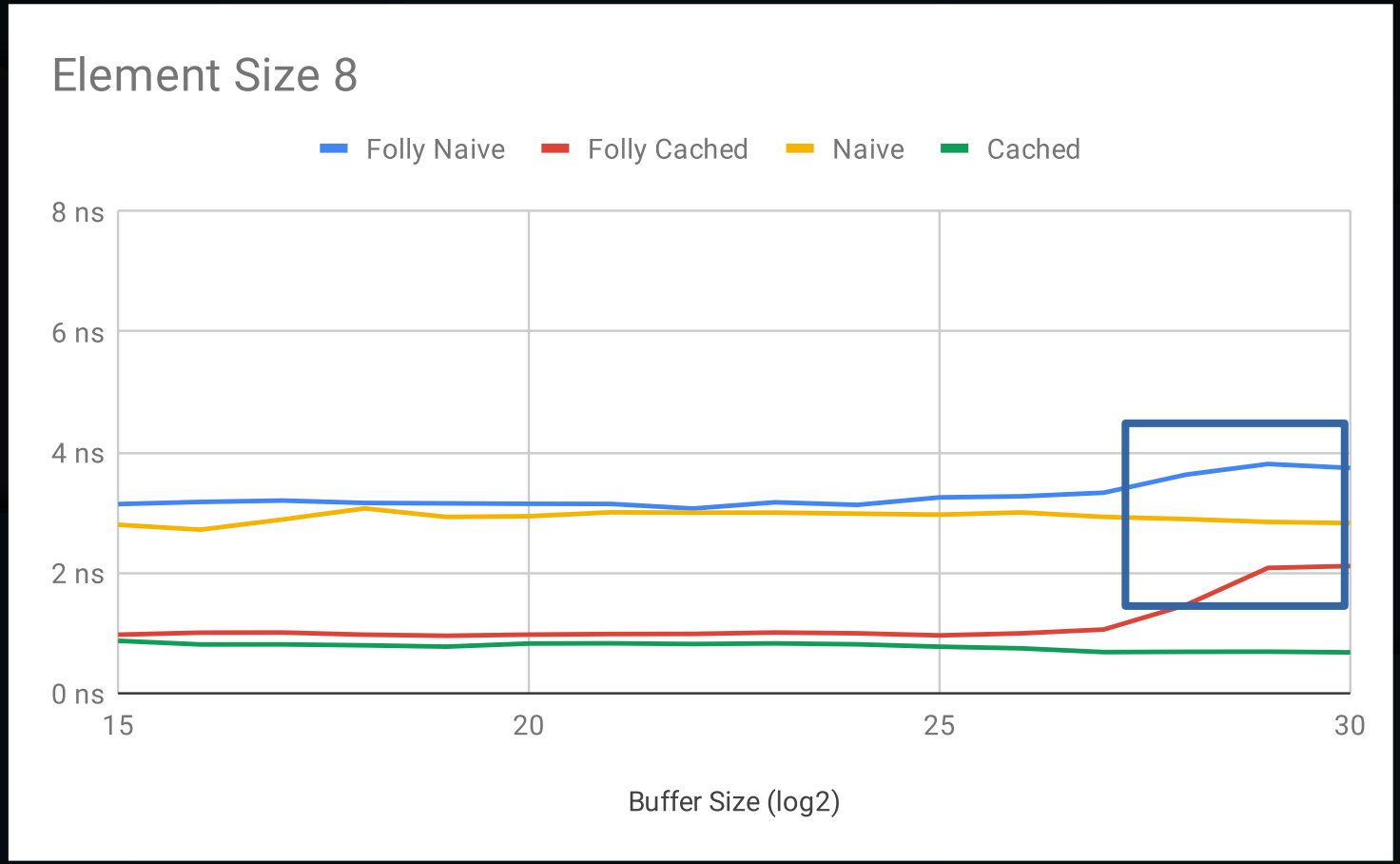
Benchmarks – Queue

- No overhead per element
- Elements are initialized

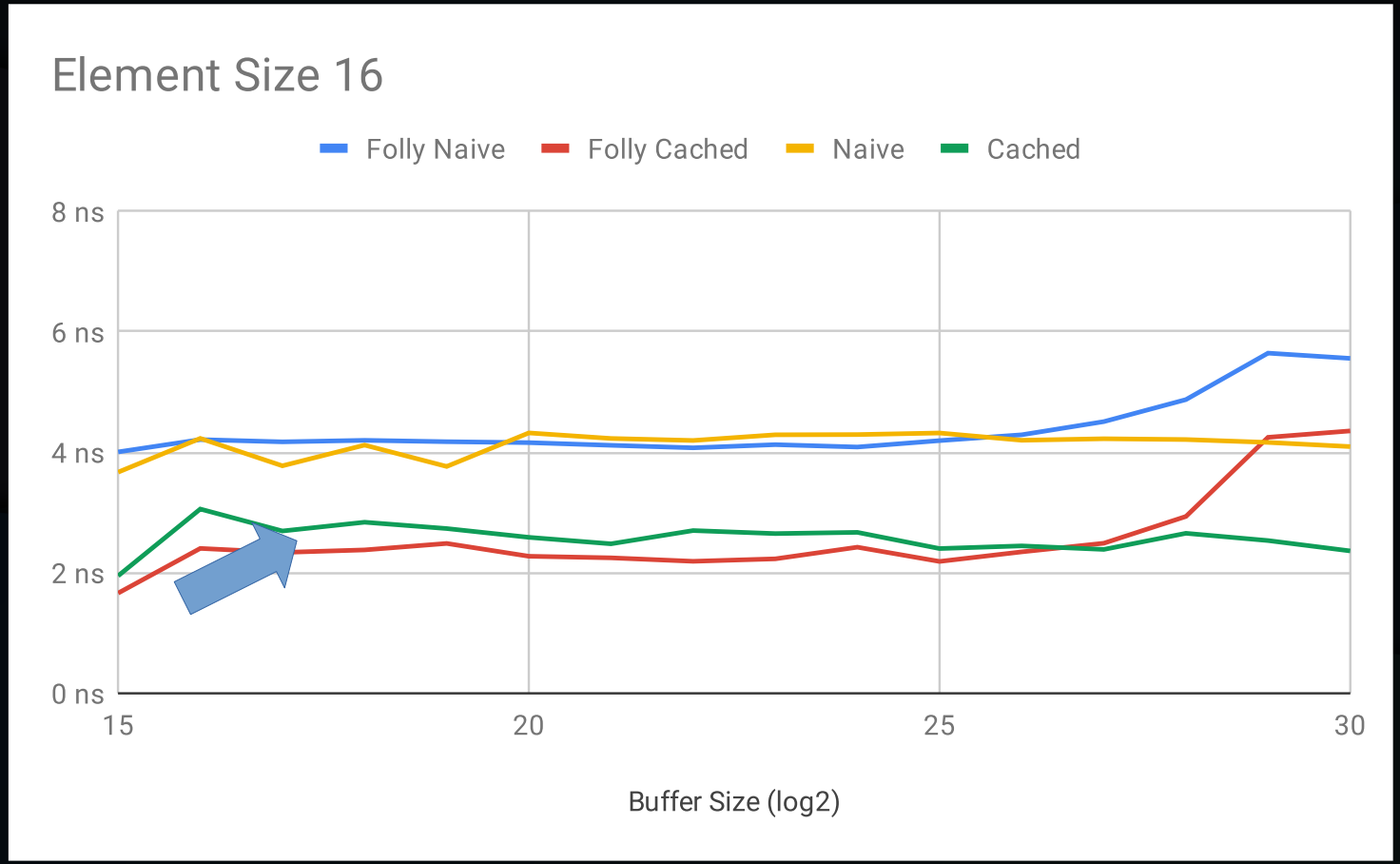
Benchmarks

- folly::ProducerConsumerQueue
  - Created by Facebook
  - Cached version by me

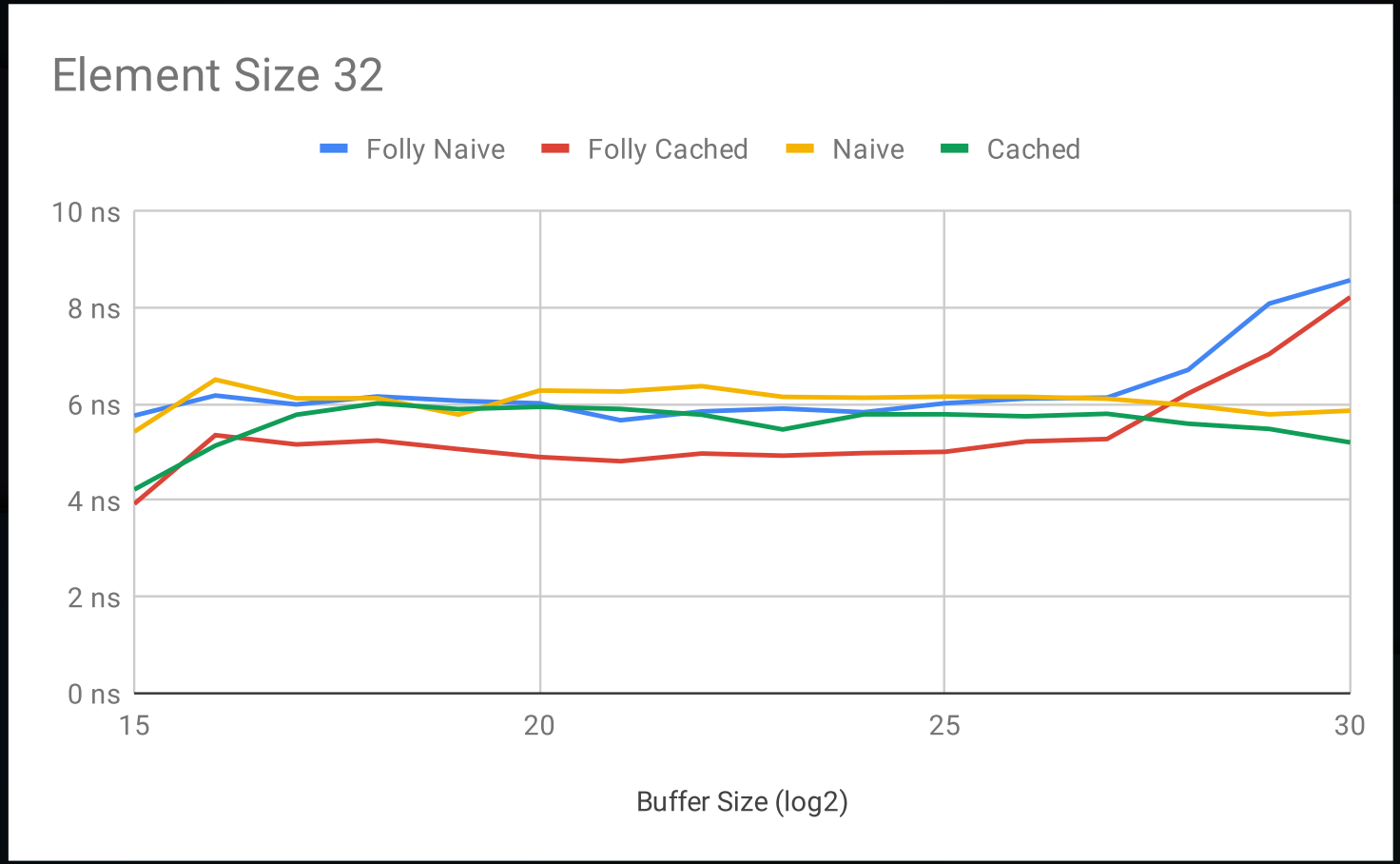
Benchmarks


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

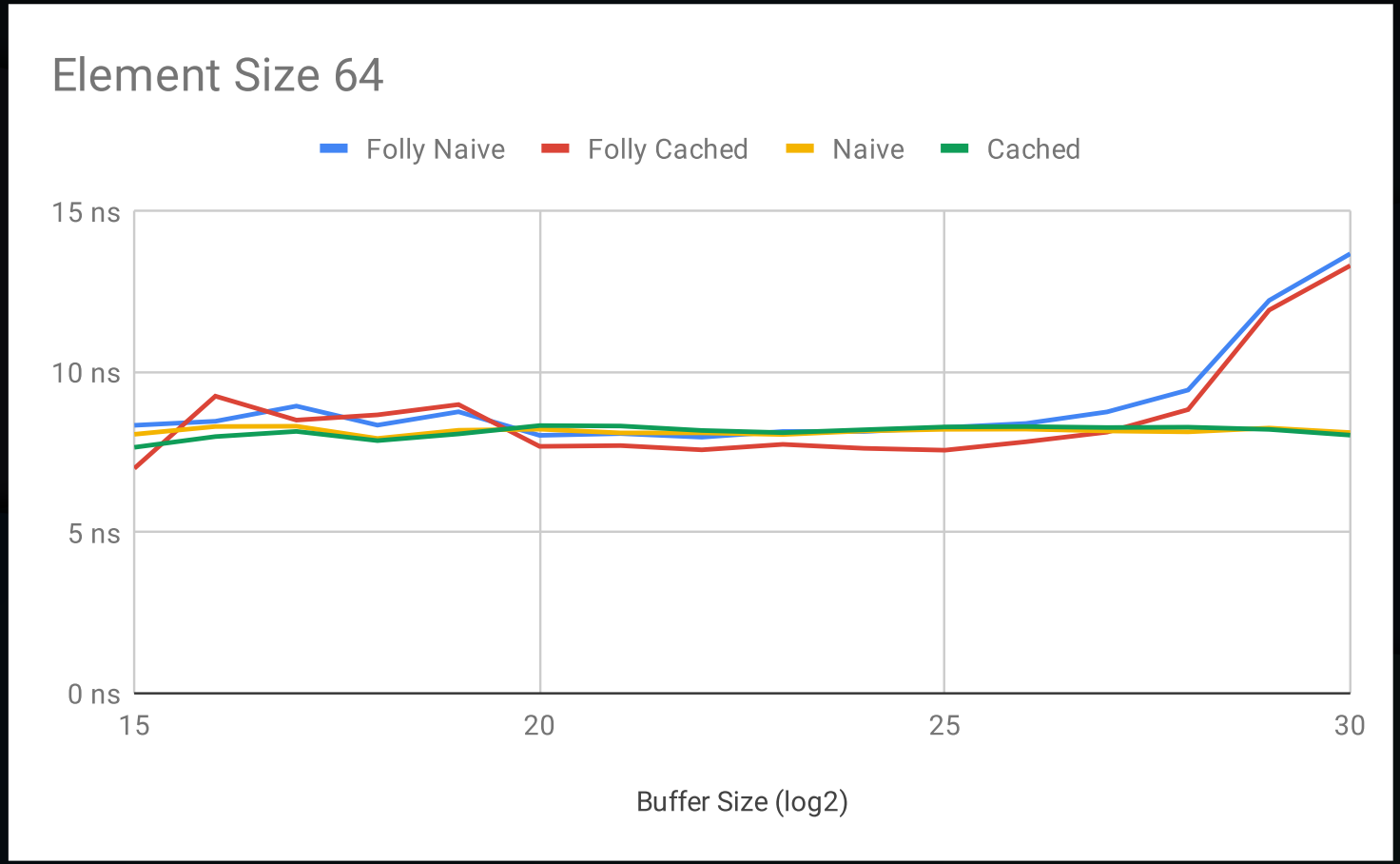
Benchmarks


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

Benchmarks


Element Size 32
Folly Naive
Folly Cached
Naive
Cached
10 ns
8 ns
6 ns
4 ns
2 ns
0 ns
15
20
25
30
Buffer Size (log2)

Benchmarks


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

Conclusion

- Optimization is viable, though effect negligible with larger element sizes
- Emplace-style interface probably best for Queues that arent larger than 128MiB
- Consume interface not that crucial, personally prefer interface with Callable, but larger code size

Questions?

Links

Ring Buffer Benchmark Results

Queue Benchmark Results

GitHub Repository

Folly implementation

Boost implementation

As an Aside

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