Instrumentation at Scale

Having Your Performance Cake and Eating it Too

Brian Martin (he/they) - QCon SF 2025

Brian Martin (he/they)

- Co-Founder @ IOP Systems
- Infrastructure, optimization, and performance
- Previously @ Twitter
- Rezolus, RPC-Perf, Pelikan, …

Rezolus: Systems Telemetry

- eBPF-based instrumentation for Linux
- Collects CPU, network, disk, scheduler metrics
- Prometheus exposition
- Open-source: https://rezolus.com

Metriken: Low-overhead Metrics

- Used in Rezolus, RPC-Perf, and Pelikan
- Optimized for performance-critical paths
- Link-time registry, minimal runtime cost

eBPF?

Extended Berkeley Packet Filter

Run sandboxed programs in the Linux kernel

- Observe kernel behavior without code changes
- Attach to kernel functions, tracepoints
- Share data via BPF maps

Let’s Begin

Why Instrumentation Matters

You need visibility into your systems

Without enough instrumentation:

- Miss critical performance issues
- Can’t diagnose production incidents
- Limits ability to improve performance

Instrumentation isn’t optional

The Cost Problem

Instrumentation adds overhead

The more you instrument, the more the costs matter

Real costs

Same operation, different implementations:

- Counter: 5ns → 1us+ (200x)
- Histogram: 7ns → 10us+ (1400x difference)

What You’ll Learn

- Cost spectrum of instrumentation
- What real libraries do
- Techniques for fearless instrumentation

Metric Types

A Quick Primer

Counter

Monotonically non-decreasing value

```
requests_total.increment();
```

Gauge

Point-in-time value

```
queue_depth.set(current);
```

Histogram

Distribution of values

```
request_duration.increment(latency_ns);
```

Rust Metrics Libraries

Design Decisions

- all these libraries make design decisions
- even the fastest might not always be right for you
- but awareness of the trade-offs is critical!

metrics

- focused on flexibility
- facade pattern
- designed for libraries and applications

prometheus

- focused on Prometheus exposition
- metrics registry
- designed for applications

goodmetrics

- focused on high-cardinality metrics
- OpenTelemetry integration
- designed for applications

metriken

- focused on efficient metrics primitives
- link-time registry for static metrics
- best when vertically integrated

Let’s understand the **performance implications**.

Counter Implementations

- Thread-local **< 1ns**
- Atomic increment **~5ns**
- Compare-and-swap (CAS) **~10ns**

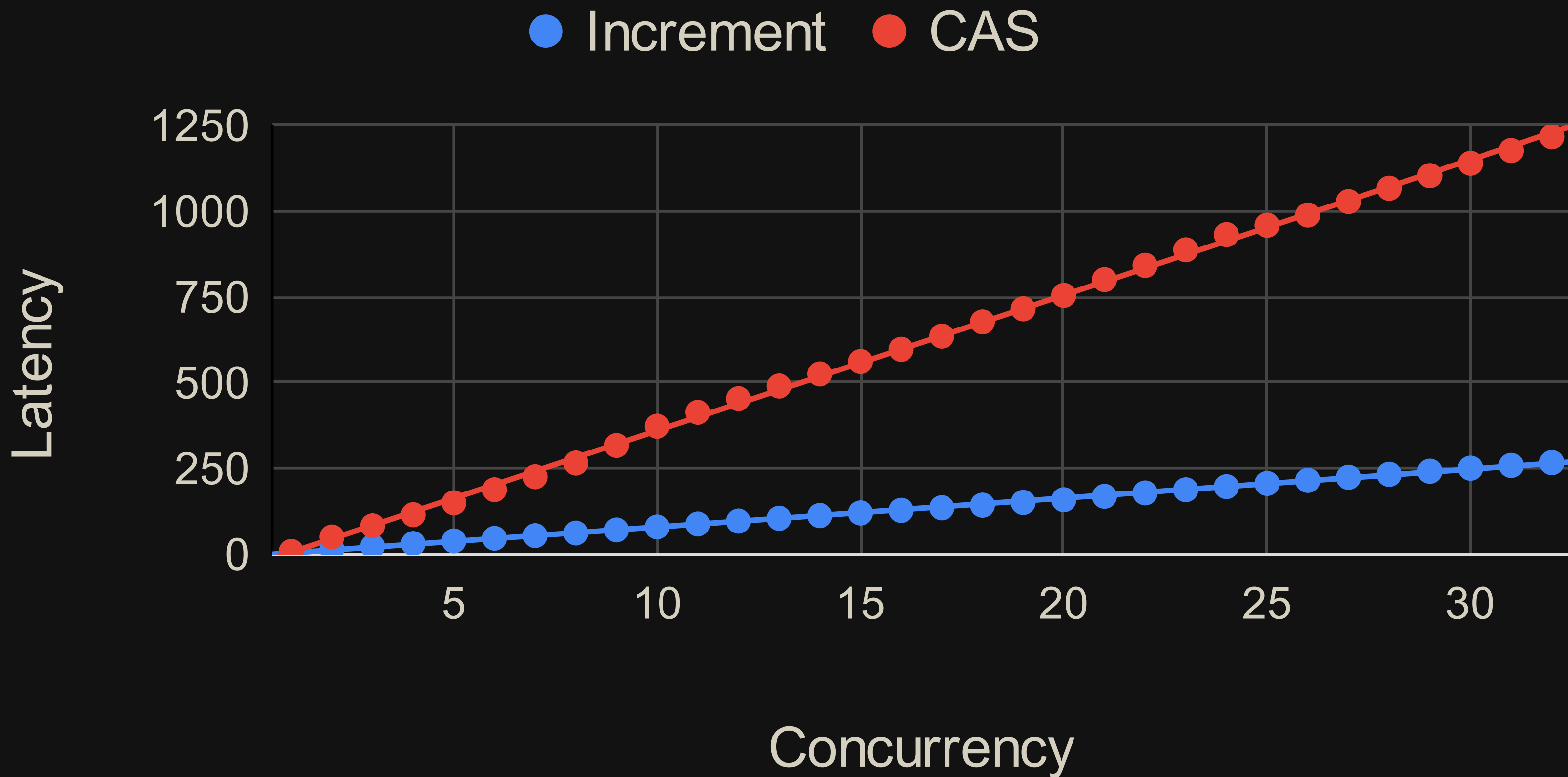
Contention

Multiple threads updating the same memory location

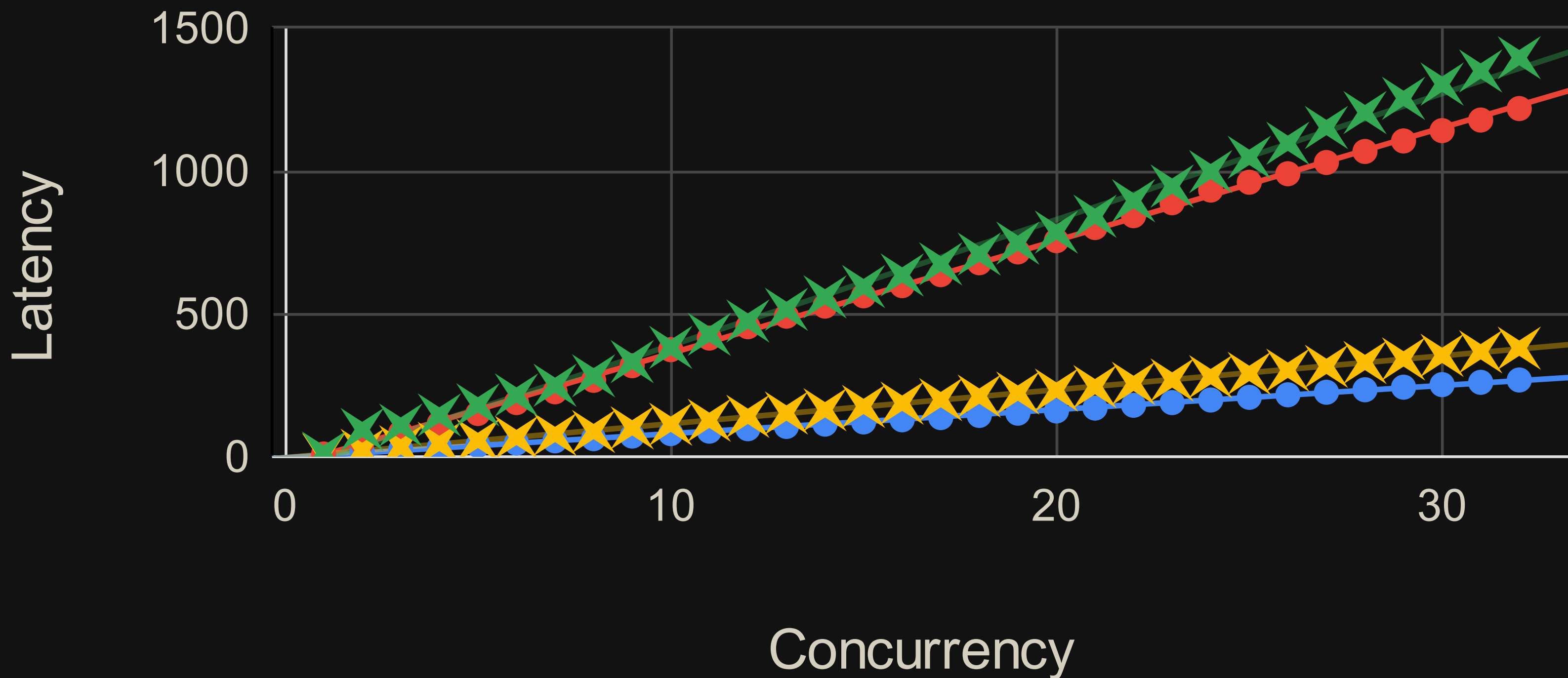
What drives the cost:

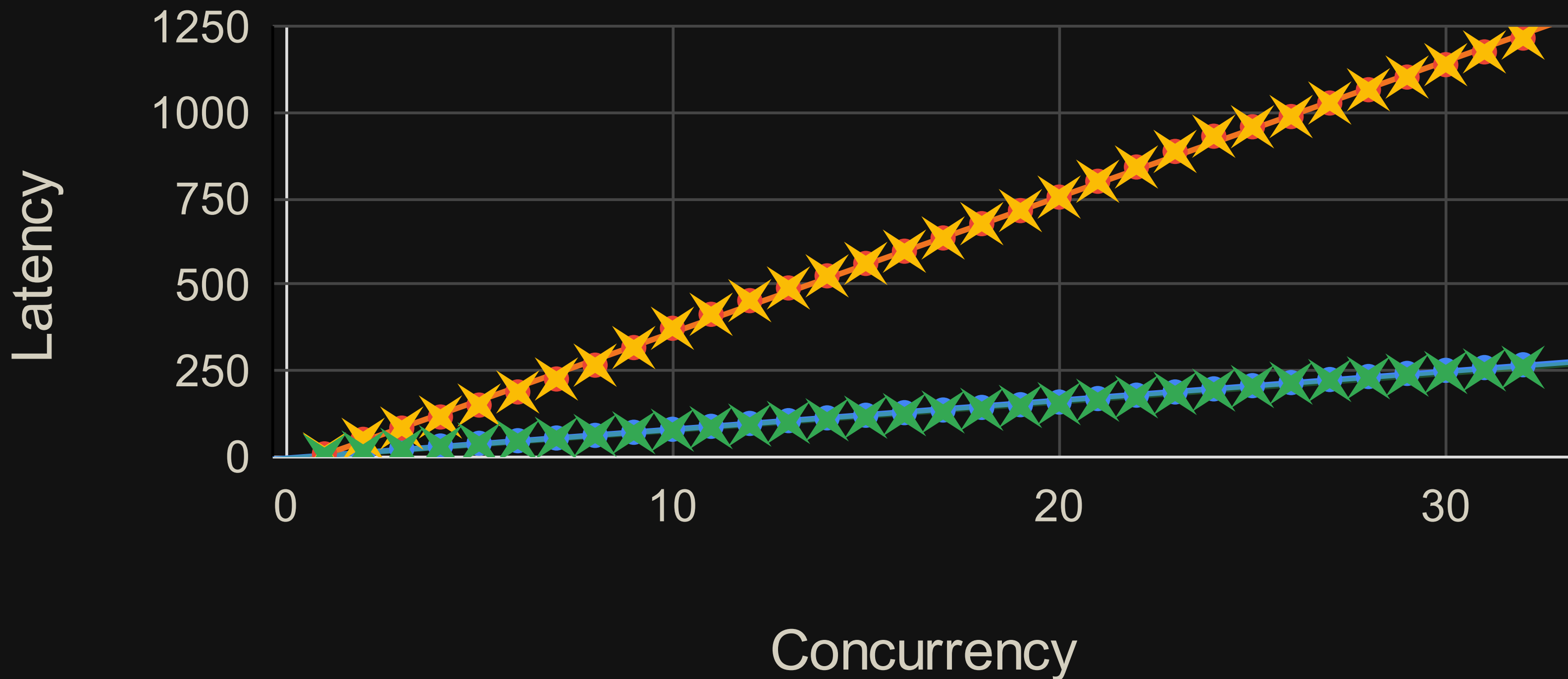
- Cache line synchronization
- Memory bus traffic
- Retry loops (for CAS operations)

Contention amplifies implementation differences

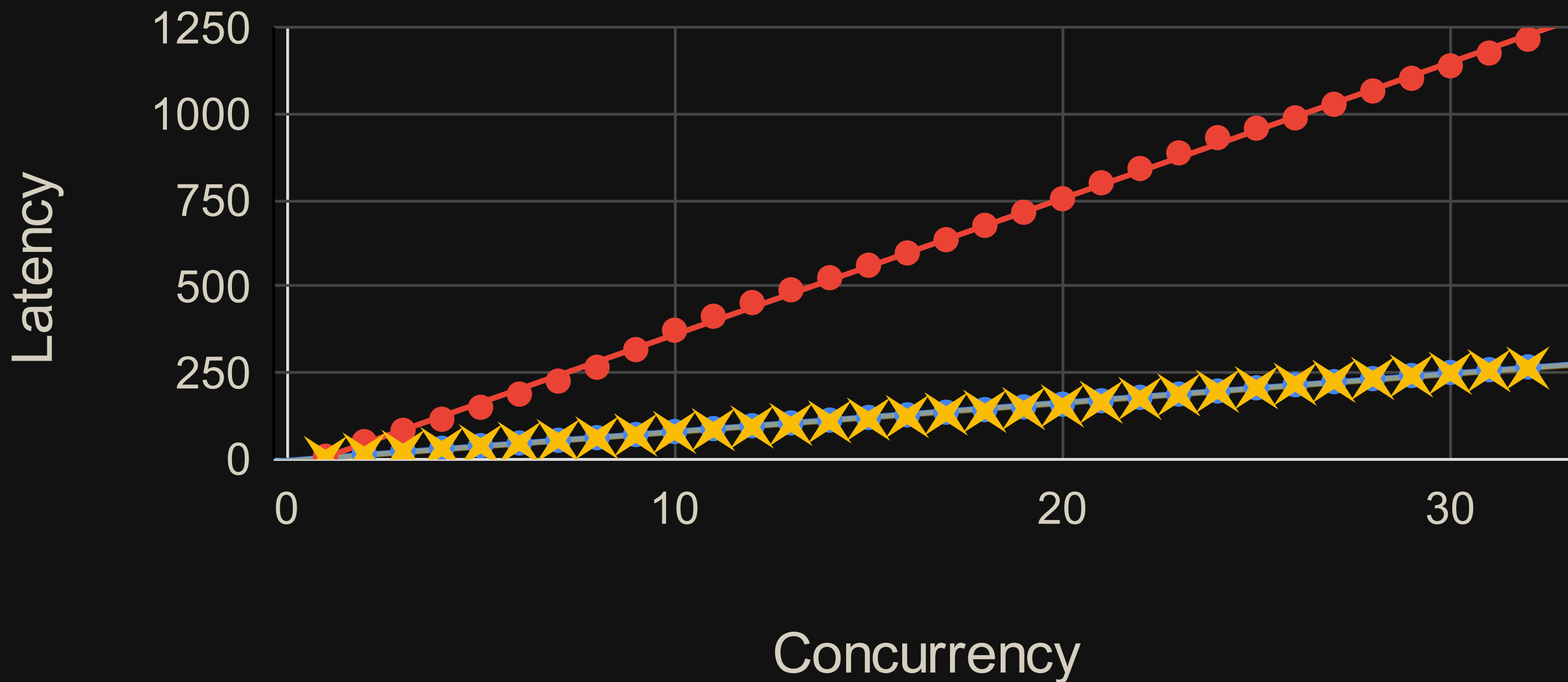

Counter Contention
Increment
CAS
Latency
Concurrency
1250
1000
750
500
250
0
5
10
15
20
25
30

Counter Contention: metrics

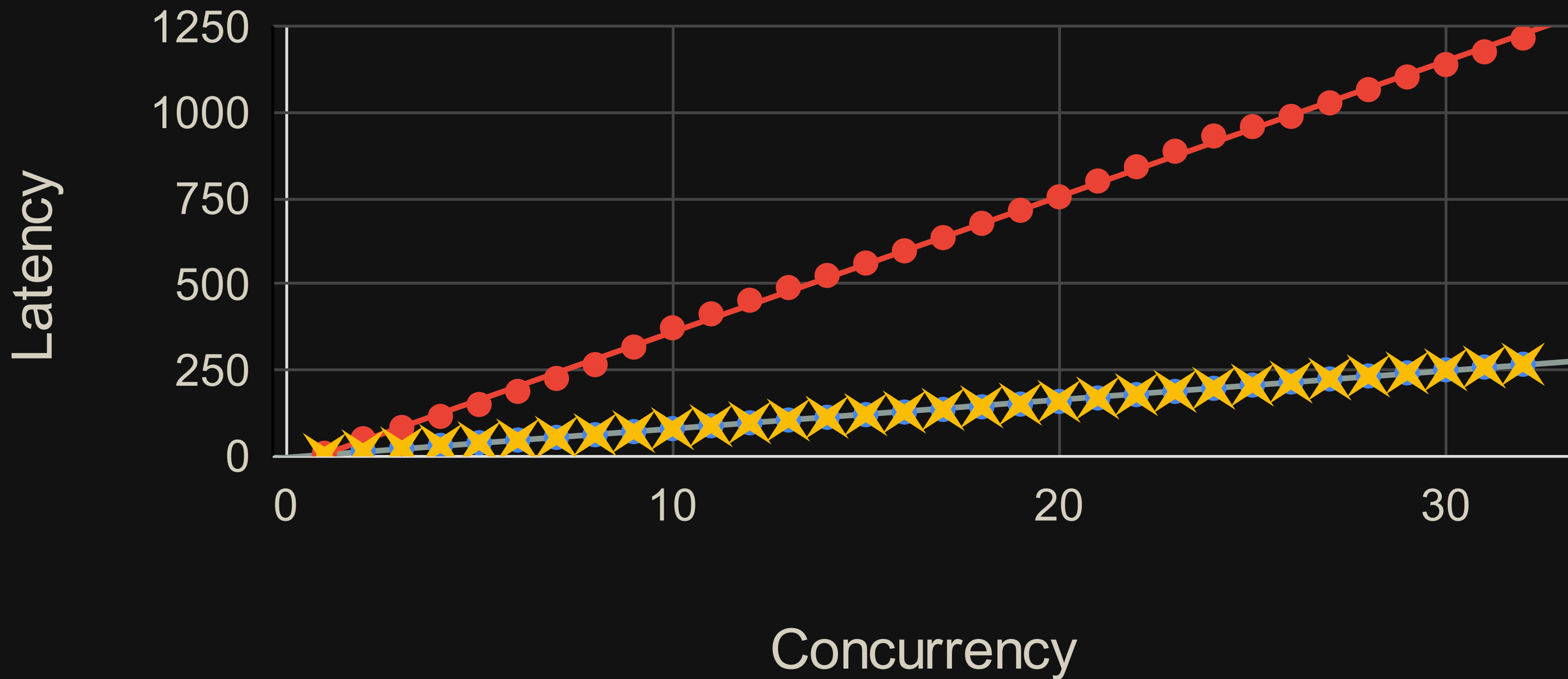



Counter Contention: Prometheus
Increment
CAS
Counter
IntCounter
Latency
1250
1000
750
500
250
0
0
10
20
30
Concurrency

Counter Contention



Counter Contention



Counter Performance

Library	1 core	8 cores	32 cores
metrics: debug	7.1 ns	90 ns	380 ns
metrics: prometheus	7.1 ns	280 ns	1400 ns
prometheus: IntCounter	5.0 ns	65 ns	270 ns
prometheus: Counter	8.8 ns	270 ns	1200 ns
goodmetrics	5.0 ns	65 ns	270 ns
metriken	5.0 ns	65 ns	270 ns

Real-World Impact

Scenario: total_requests on 32-core system

Implementation	Contended Cost	Max Throughput
metriken	270ns	**119M req/s**
metrics: prometheus	1400ns	**23M req/s**

~5x difference in system ceiling

Can We Do Better?

The Contention Problem

Single shared counter = cache line bouncing

All cores compete for the same memory location

5ns → 270ns under contention (54x slower)

Per-CPU Counters

Give each CPU its own counter

```
counter[idx].increment();  // No contention!
```

Read by summing across all CPUs

```
total = counter.iter().sum();
```

Used extensively in Linux kernel for hot paths

False Sharing

Problem: Adjacent counters share cache lines

Cache line is 64 bytes = 8 × u64 counters

CPU 0 updates `counter[0]`

CPU 1 updates `counter[1]`

Still fighting over the same cache line!

Counter Groups

Group related counters per CPU

```
struct CounterGroup {
    requests: u64,
    errors: u64,
    bytes: u64,
    // ... up to 8 counters (1 cache line)
}
counters: [CounterGroup; NUM_CPUS]
```

Each CPU has its own cache line(s)

Counter Groups in eBPF

```
#define CACHE_LINE 8 // bytes
#define CPUS 1024    // max CPUs

struct {
    __uint(type, BPF_MAP_TYPE_ARRAY);
    __uint(map_flags, BPF_F_MMAPABLE);
    __uint(max_entries, CPUS * CACHE_LINE);
} counters SEC(".maps");
```

Same technique - each CPU gets 8 contiguous counters

Counter Approaches

3 main approaches

Implementation	Contended Cost	Max Throughput
CAS loops	1200 ns	**~27M req/s**
Atomic Add	270 ns	**119M req/s**
Sharded	5 ns	**6.4B req/s**

Approach makes a huge difference!

Gauges

Histograms

Histogram Structure

Array of buckets counting values in ranges

```
struct Histogram {
    buckets: [u64; N],  // Counts per bucket
    // bucket 0: 0-10ns
    // bucket 1: 10-100ns
    // bucket 2: 100-1000ns
    // ...
}
```

Core operation: given a value, which bucket?

Indexing

Stragey 1: Linear Search

- Walk array comparing to boundary values
- O(n) - simple but slow with many buckets

Stragey 2: Binary Search

- Binary search through boundaries
- O(log n) - better, ~6 comparisons for 50 buckets

Strategy 3: Direct Indexing

- Compute bucket index directly from value
- O(1) - fastest, requires mathematical bucketing scheme

Direct Indexing: Linear Ranges

```
idx = value / width;  // Fast indexing!
```

Problems:

- bucket count vs resolution
- large relative error at low end

Direct Indexing: Log Buckets

```
idx = log10(value);  // Better relative error
```

Problems:

- logarithms are slow
- large relative error

Direct Indexing: Log2 Buckets

```
bucket = 64 - value.leading_zeros(); // fast!!!
```

Problems:

- large relative error

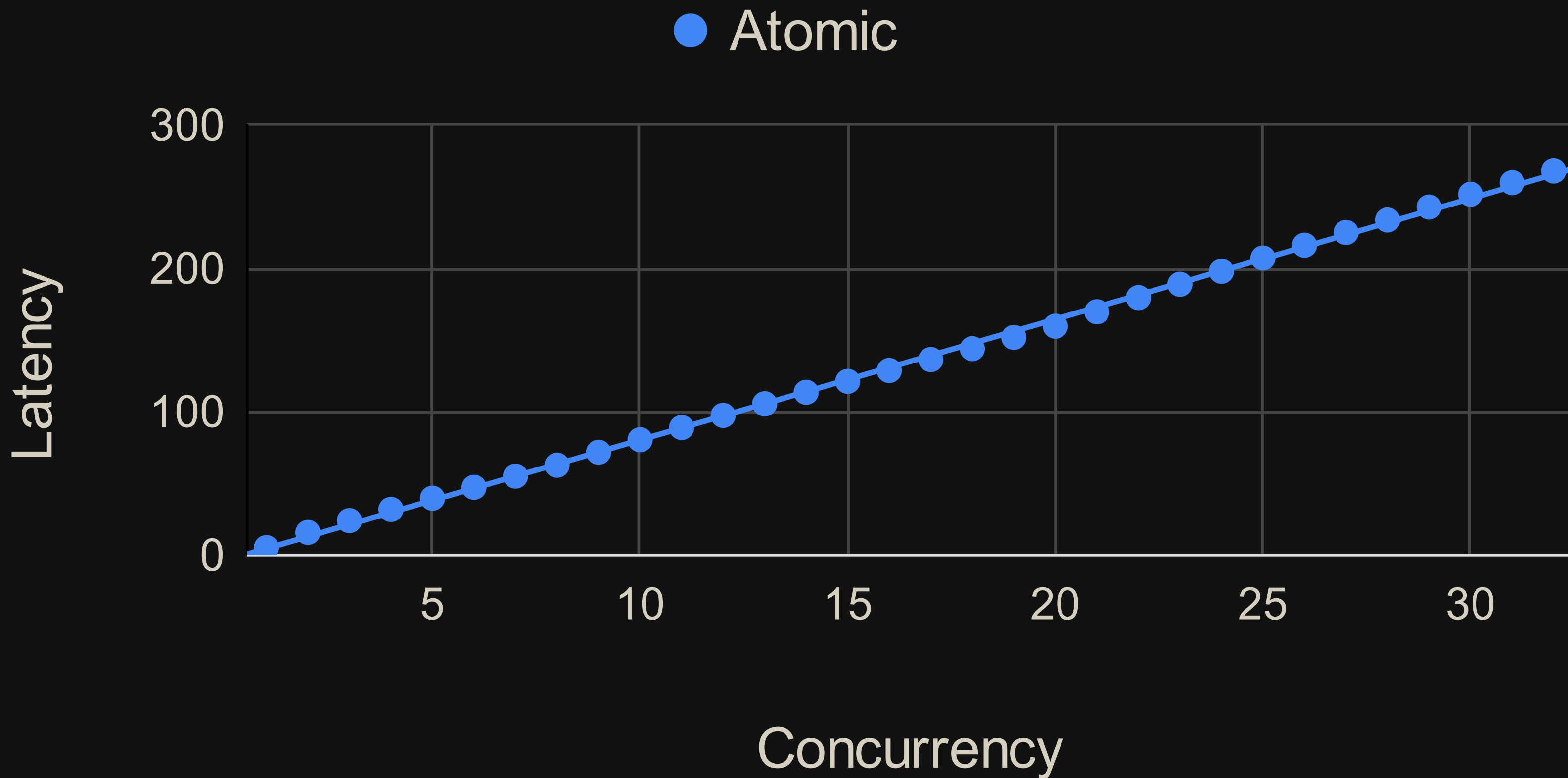
Direct Indexing: HDR / H2 Histogram

- Log2 outer buckets (very fast)
- Linear sub-buckets
- Configurable precision

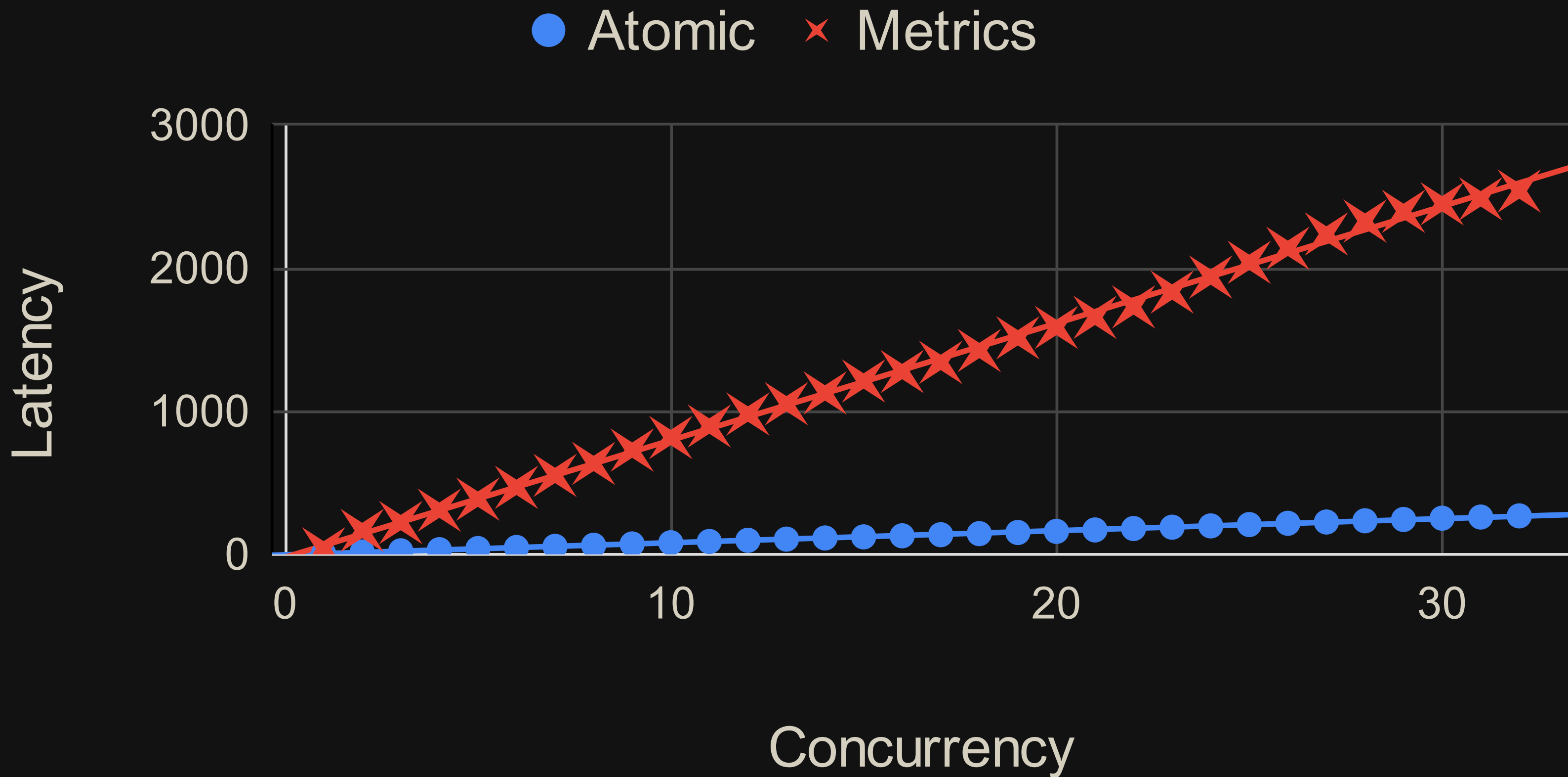
Standalone Histograms

Library	Latency
hdrhistogram	2.65 ns
histogram	2.15 ns

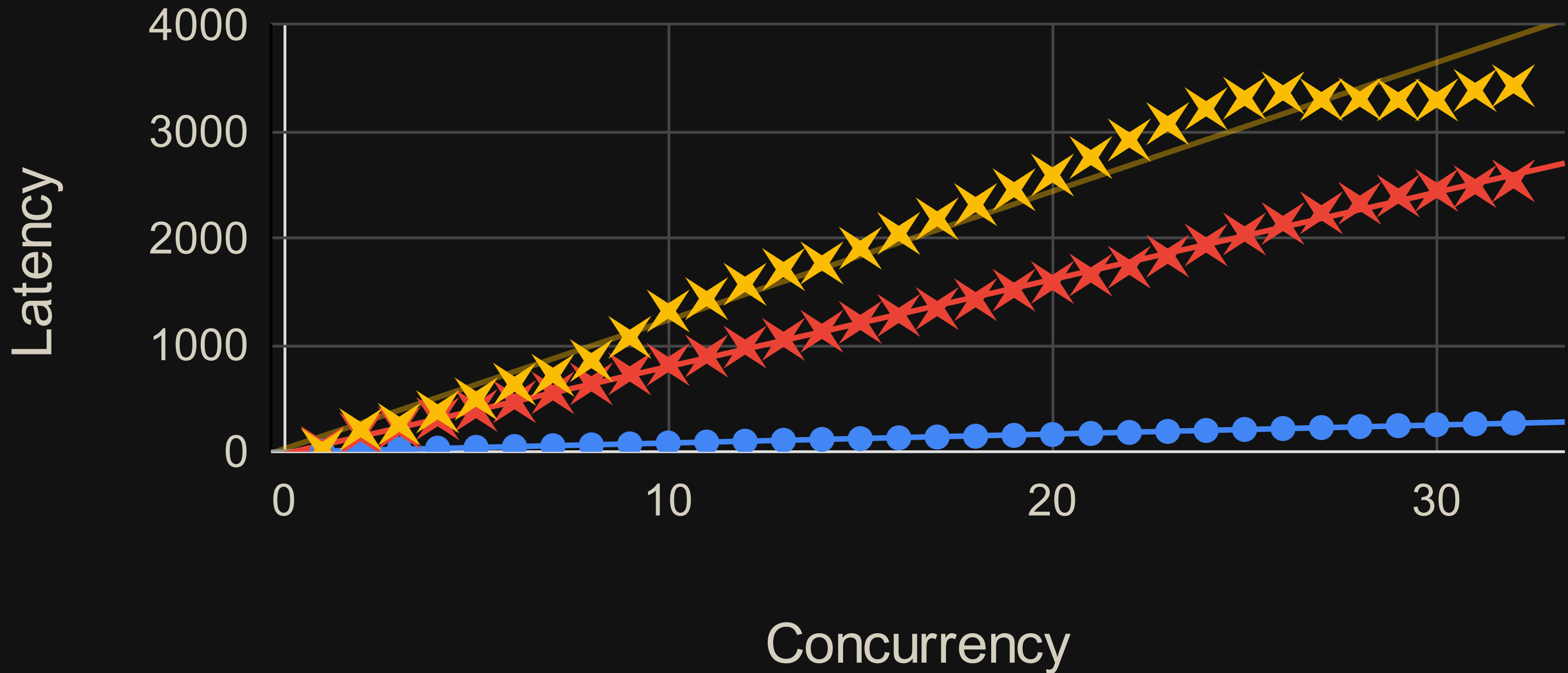
Histogram Contention:



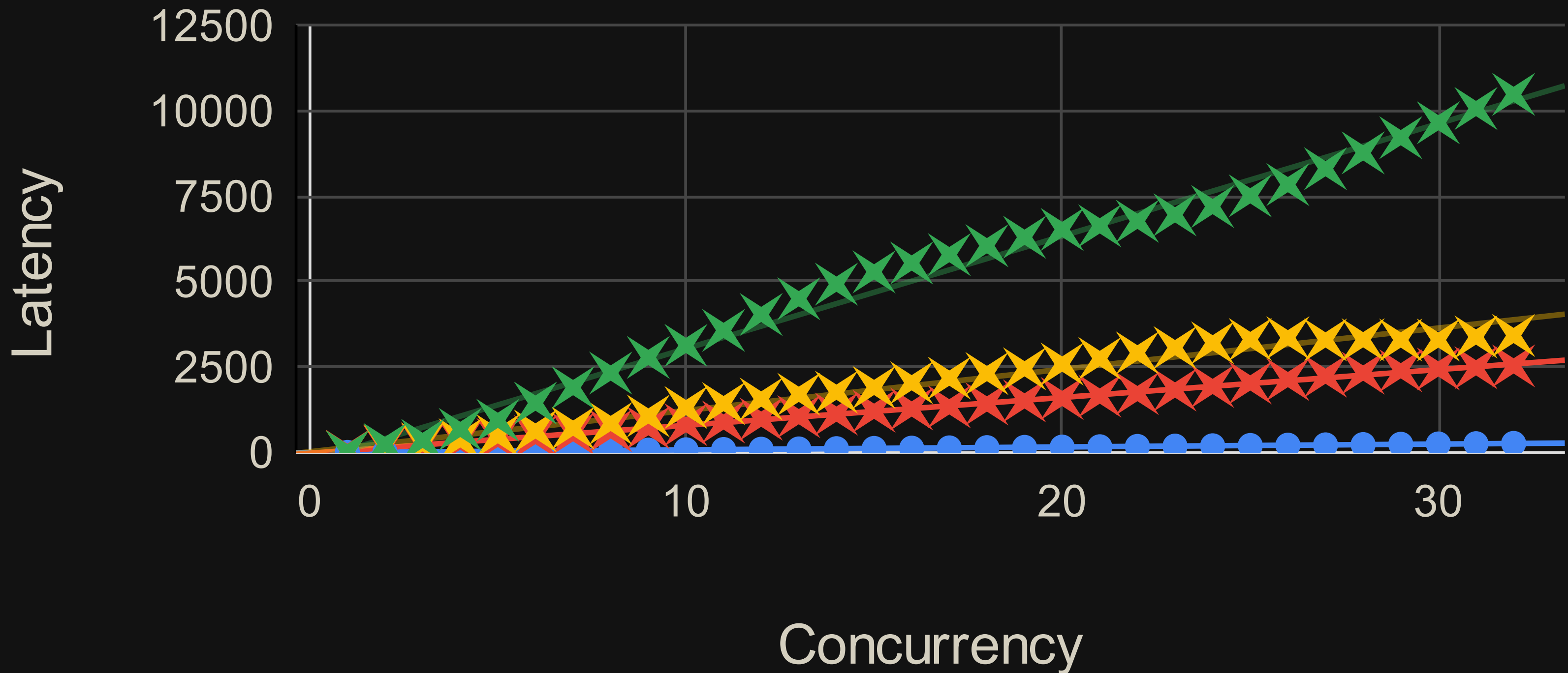
Histogram Contention:



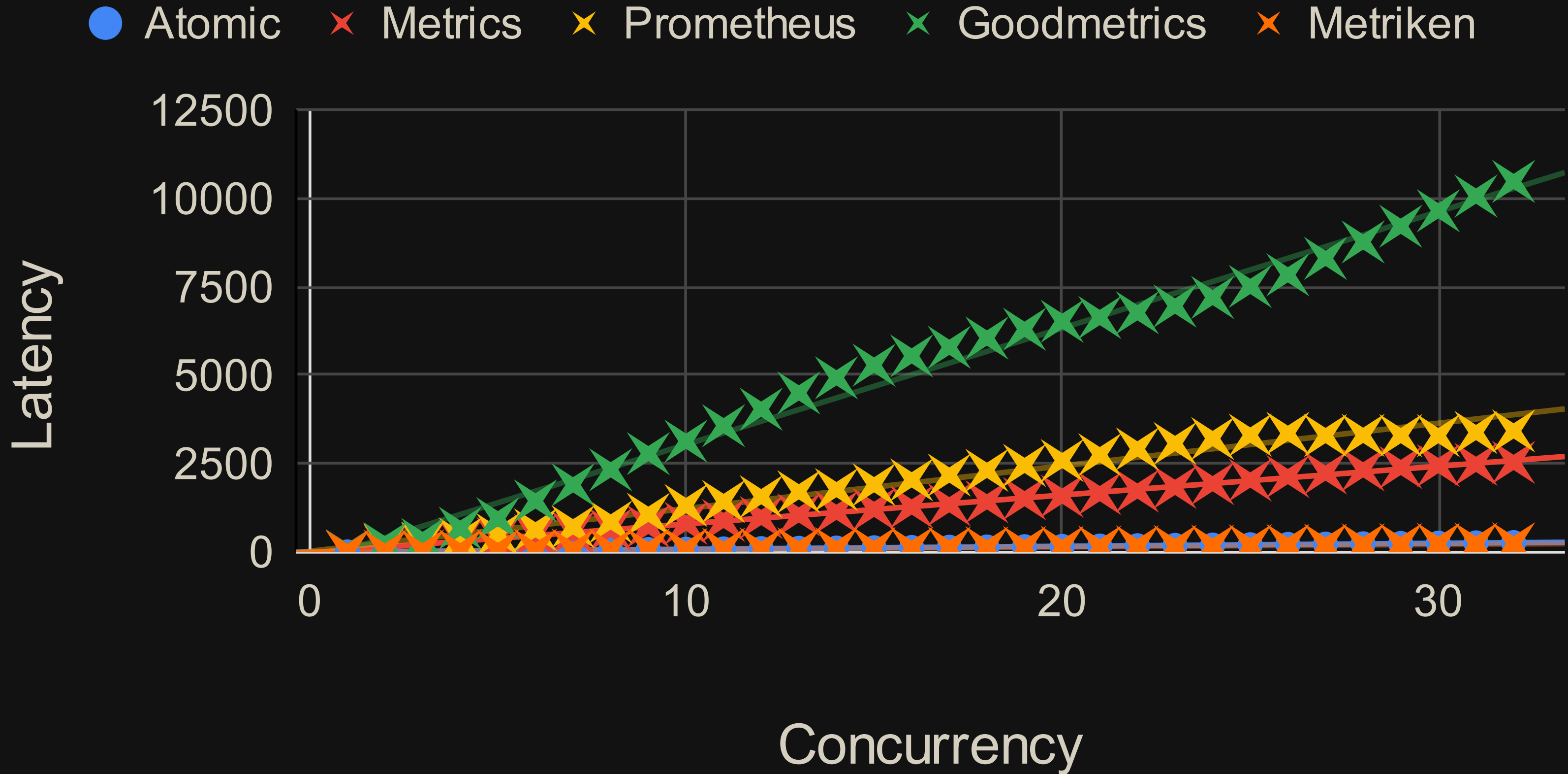
Histogram Contention:



Histogram Contention:



Histogram Contention:



Histogram Increment Summary

Library	Uncontended	Contended
metrics	38 ns	2500 ns
prometheus	20 ns	3400 ns
goodmetrics	24 ns	10500 ns
metriken	7 ns	235 ns

metrics

- walks bounds to find bucket
- crossbeam-epoch overhead
- multiple atomics (4+ per increment)

prometheus

- multiple atomics (6+ per increment)
- CAS loop for tracking sum (AtomicF64)

goodmetrics

- mutex acquisition
- serializes all updates
- simplifies implementation

metriken

- lock-free increments
- efficient direct indexing
- single atomic operation

Histogram Trade-offs

Consistent Snapshots

- requires multiple atomics or locks

Eventually Consistent

- much faster updates

H2 Histograms in eBPF

Same h2 indexing algorithm works in kernel

```
struct {
    __uint(type, BPF_MAP_TYPE_ARRAY);
    __uint(map_flags, BPF_F_MMAPABLE);
    __uint(max_entries, NUM_BUCKETS);
} hist SEC(".maps");

u32 bucket = h2_index(latency_ns);
array_incr(&hist, bucket); // atomic add
```

What Have We Learned?

Tough Choices

- performance vs flexibility
- counters vs histograms
- strong vs eventual consistency

Design Principles

Counters

- Atomic fetch_add is most efficient
- Avoid CAS loops, prefer normal atomic primitives
- Reduce contention with sharding

Cost: 5ns → 270ns (contended)

Measured on c8g.8xlarge

Histograms

- Eventually consistent is most efficient
- Can be nearly as cheap as a counter
- Implementation matters greatly

Cost: 7ns → 235ns (contended)

Measured on c8g.8xlarge

Some implementations are much worse!!!

Key Takeaways

Implementation Matters

Same operation, 4-40x difference:

Counters: 270ns → 1200ns (atomic vs CAS)

Histograms: 235ns → 10,500ns (atomic vs mutex)

The difference between:

- Fearless instrumentation
- Rationing every metric

Fearless Instrumentation

With the right primitives:

- Atomic increments, not CAS loops
- Direct indexing, not searches
- Per-CPU sharding to eliminate contention
- Same techniques work in userspace and eBPF

You can instrument comprehensively without fear

Thanks!

Resources:

- Rezolus: https://rezolus.com
- Metriken: https://github.com/iopsystems/metriken