High Performance Erlang: Pitfalls and Solutions

MZSPEED Team

Machine Zone, Inc.

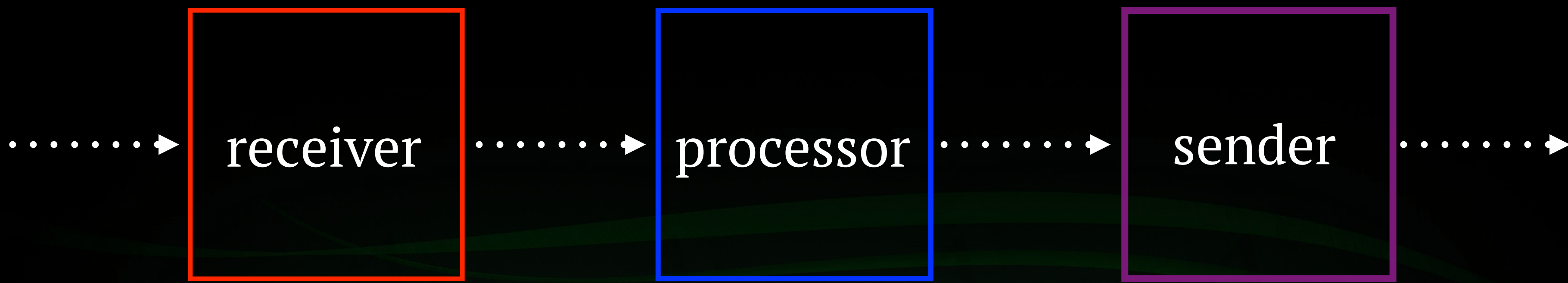
Agenda

- Erlang at Machine Zone
- Issues and Roadblocks
- Lessons Learned
- Profiling Techniques
- Case Study: Counters

Erlang at Machine Zone

- High performance data processing system
- How to make it fast?

Typical Approach: Pipelined tasks



Execute different logic in different processes

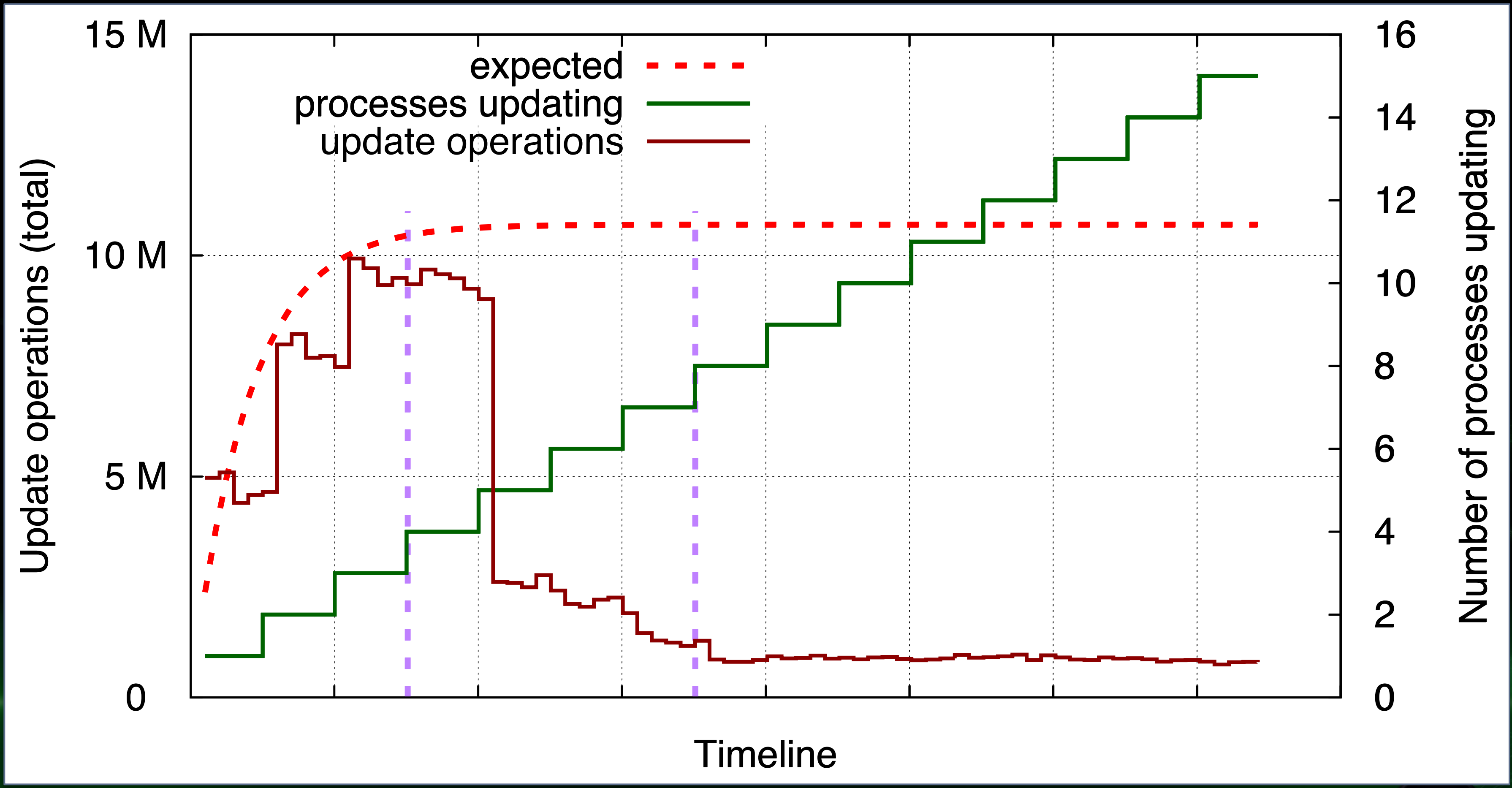
How do we exchange data between processes?

How do we make it fast?

ETS

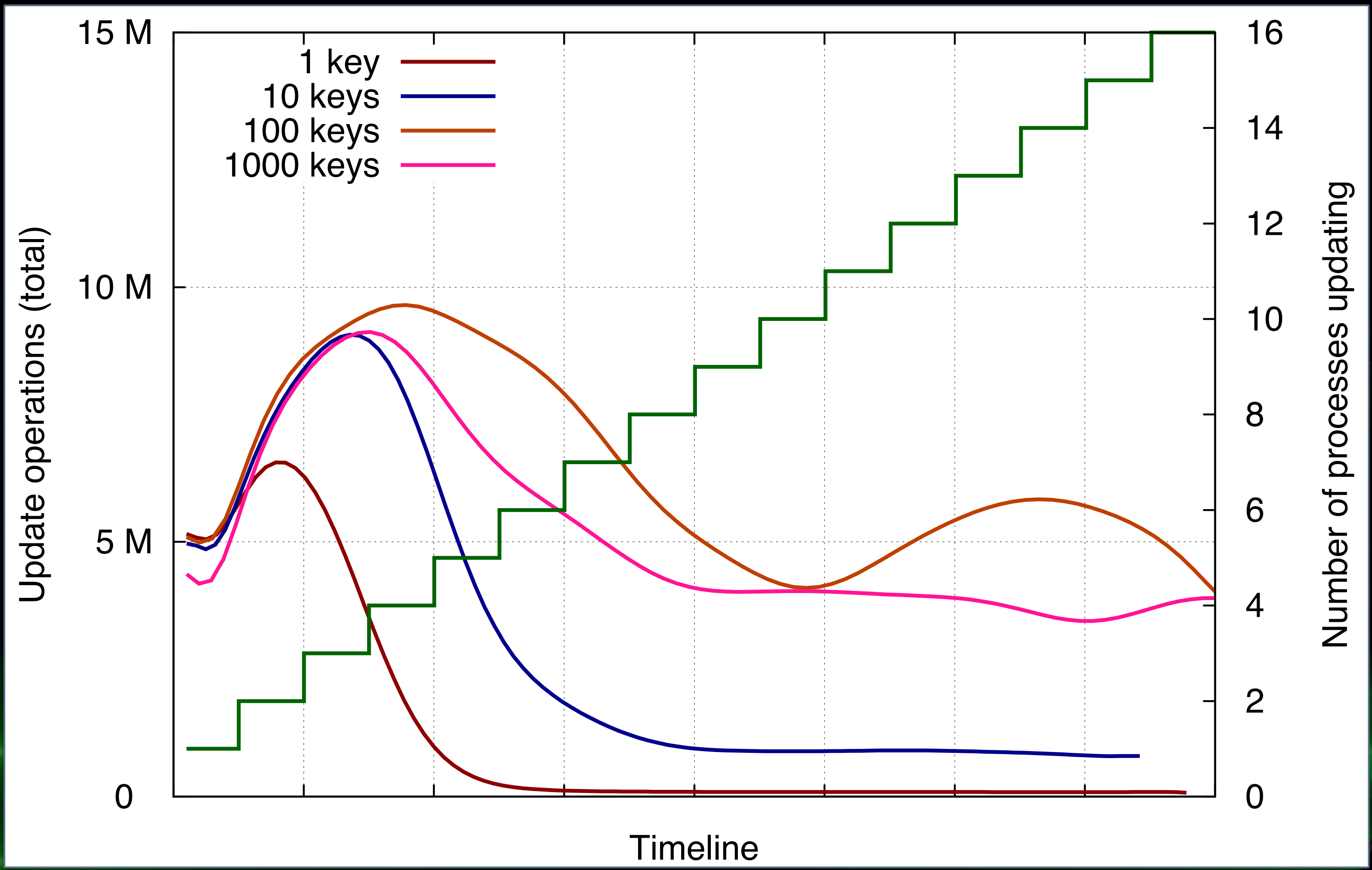
ETS: update_counter, 10 Keys: Expected vs Observed Performance

- Observed Performance is far worse than Expected



ETS: update_counter, Multiple Keys: Observed Performance

- Performance is degraded even with multiple keys



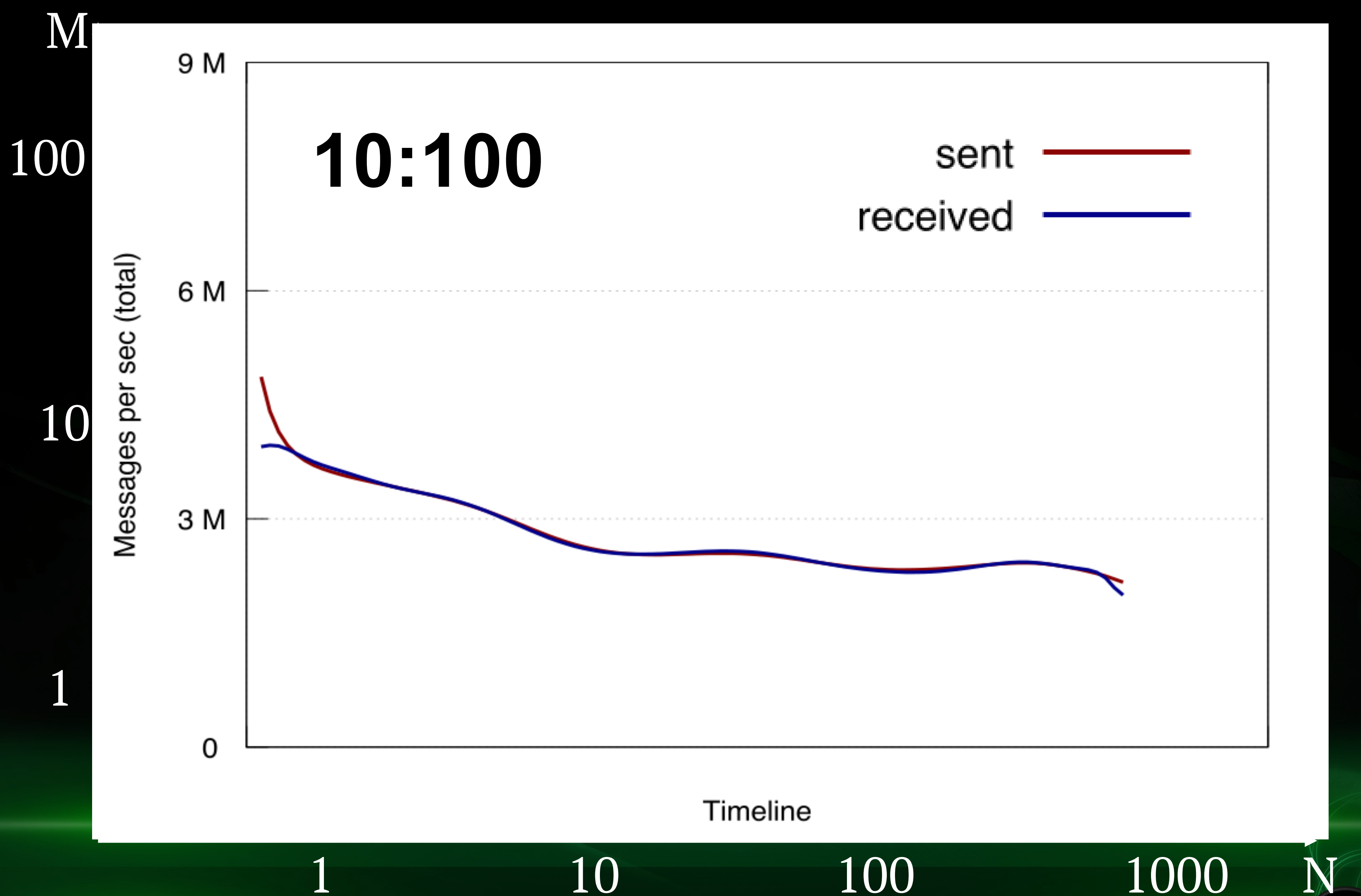
ETS has Locks

- Fine-grained locks
- 64 locks per table by default

```
Thread 5 (Thread 0x7efc4f2ba700 (LWP 14494)):
#0  0x00007efc61cbb199 in syscall () from /lib64/libc.so.6
#1  0x0000000000692533 in wait__ ()
#2  0x0000000000692622 in ethr_event_swait ()
#3  0x000000000068f4e2 in event_wait ()
#4  0x000000000068f82f in write_lock_wait ()
#5  0x00000000006908bf in rwmutex_normal_rwlock_wait ()
#6  0x000000000069158e in ethr_rwmutex_rwlock ()
#7  0x000000000055befd in erts_rwmtx_rwlock ()
#8  0x000000000055bfae in erts_smp_rwmtx_rwlock ()
#9  0x000000000055c6e8 in WLOCK_HASH ()
#10 0x00000000005638f5 in db_lookup_dbterm_hash ()
#11 0x00000000005406cb in do_update_counter ()
#12 0x0000000000540f89 in ets_update_counter_4 ()
#13 0x000000000043dc74 in process_main ()
#14 0x0000000000506cd5 in sched_thread_func ()
#15 0x00000000006917ea in thr_wrapper ()
#16 0x00007efc62179a51 in start_thread () from /lib64/libpthread.so.0
#17 0x00007efc61cbe93d in clone () from /lib64/libc.so.6
```

Message Passing

Message Passing: M Processes to N Processes



Message Passing: Scheduling Locks

- Add to runq with locks

```
Thread 6 (Thread 0x7f29b667b700 (LWP 7710)):
#0  0x00007f29c8d532e4 in __lll_lock_wait () from /lib64/libpthread.so.0
#1  0x00007f29c8d4e588 in _L_lock_854 () from /lib64/libpthread.so.0
#2  0x00007f29c8d4e457 in pthread_mutex_lock () from /lib64/libpthread.so.0
#3  0x00000000004f83ff in ethr_mutex_lock ()
#4  0x00000000004f85d8 in erts_mtx_lock ()
#5  0x00000000004f894c in erts_smp_mtx_lock ()
#6  0x00000000004f9bec in erts_smp_runq_lock ()
#7  0x000000000050501b in add2runq ()
#8  0x000000000050525f in schedule_process ()
#9  0x000000000050528e in erts_schedule_process ()
#10 0x00000000004f4e47 in erts_proc_notify_new_message ()
#11 0x00000000004f5f90 in queue_message ()
#12 0x00000000004f6a7b in erts_send_message ()
#13 0x00000000004d5527 in do_send ()
#14 0x00000000004d6374 in erl_send ()
#15 0x000000000043b501 in process_main ()
#16 0x0000000000506cd5 in sched_thread_func ()
#17 0x00000000006917ea in thr_wrapper ()
#18 0x00007f29c8d4ca51 in start_thread () from /lib64/libpthread.so.0
#19 0x00007f29c889193d in clone () from /lib64/libc.so.6
```

Message Passing: Scheduler Communication Locks

- Task stealing with locks

```
Thread 7 (Thread 0x7fd94d43c700 (LWP 10061)):
#0  0x00007fd95f2ee2e4 in __lll_lock_wait () from /lib64/libpthread.so.0
#1  0x00007fd95f2e9588 in _L_lock_854 () from /lib64/libpthread.so.0
#2  0x00007fd95f2e9457 in pthread_mutex_lock () from /lib64/libpthread.so.0
#3  0x00000000004f83ff in ethr_mutex_lock ()
#4  0x00000000004f85d8 in erts_mtx_lock ()
#5  0x00000000004f894c in erts_smp_mtx_lock ()
#6  0x00000000004f9bec in erts_smp_runq_lock ()
#7  0x00000000004ffeef in try_steal_task_from_victim ()
#8  0x00000000005001bb in check_possible_steal_victim ()
#9  0x00000000005002fa in try_steal_task ()
#10 0x0000000000508dc9 in schedule ()
#11 0x0000000000434c7c in process_main ()
#12 0x0000000000506cd5 in sched_thread_func ()
#13 0x00000000006917ea in thr_wrapper ()
#14 0x00007fd95f2e7a51 in start_thread () from /lib64/libpthread.so.0
#15 0x00007fd95ee2c93d in clone () from /lib64/libc.so.6
```

Message Passing

- Limit on #messages delivered per second
  - < 10M msgs/sec on MacBook Pro
  - < 22M msgs/sec on a powerful 56-core server
- Independent of #processes

What if we want to achieve more?

Solution 1: Minimize Message Exchange

Do everything in the same process and shard

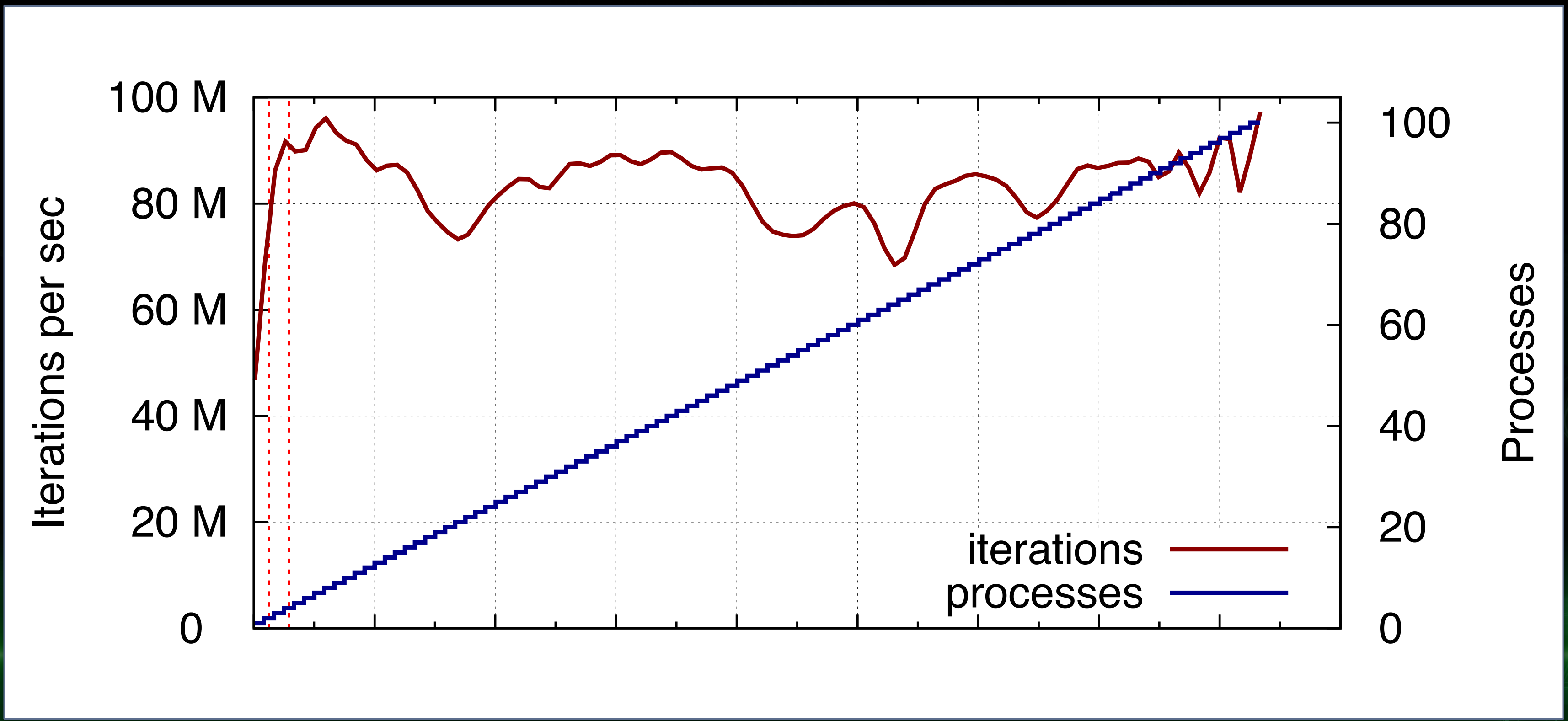
receiver
processor
sender

receiver
processor
sender

receiver
processor
sender

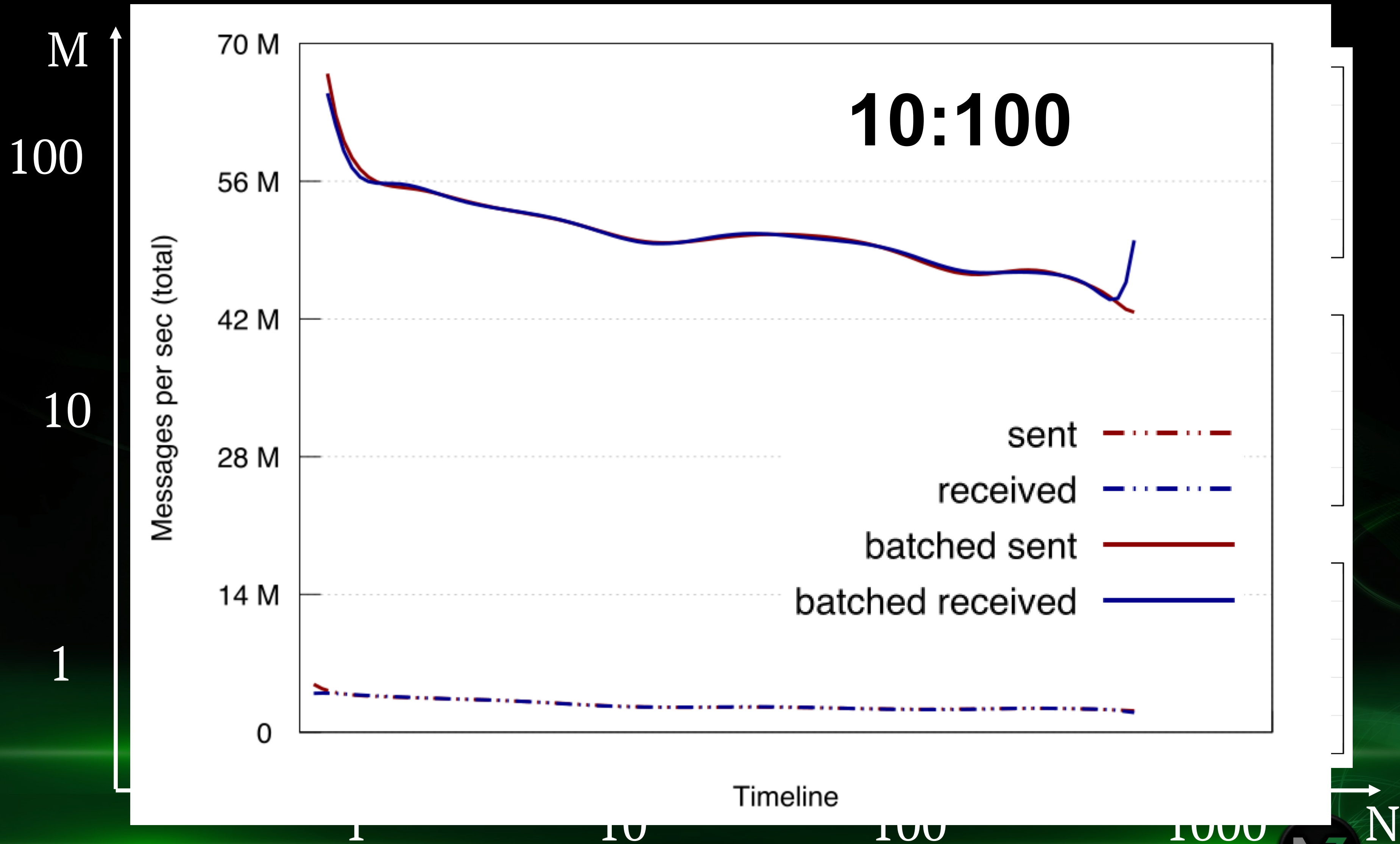
Solution 1 Speed

- 100x the #ETS updates
- 20x–30x the #message exchanges
- No decrease with increase in #processes



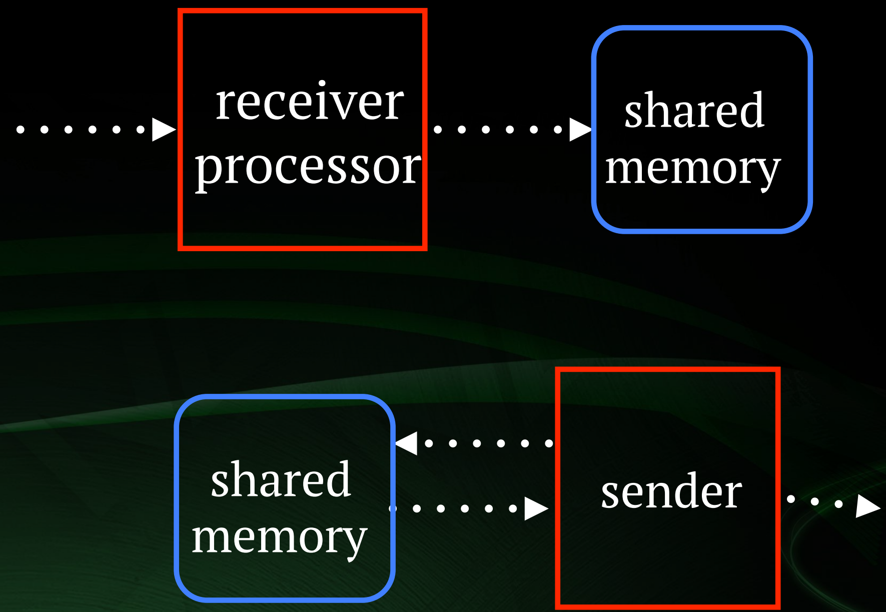
Solution 2: Message Batching

Batch messages: 5x–30x more msgs/sec



Solution 3: Domain-Specific Optimizations

- Deliver one message to multiple recipients:
  1. Receive, process, store to shared memory using NIF
  2. Read from the shared memory, send using push notification

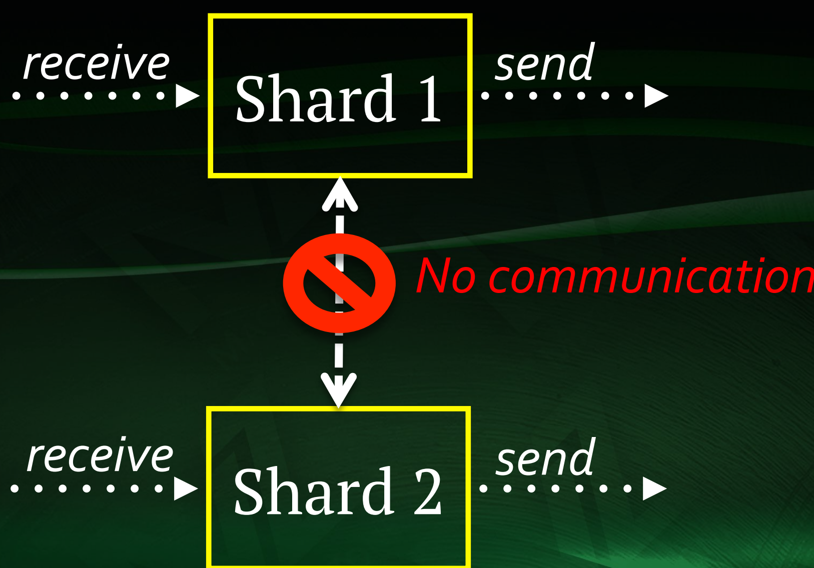


Total throughput of 600M msgs/sec to many recipients

High Performance Erlang: Lessons Learned

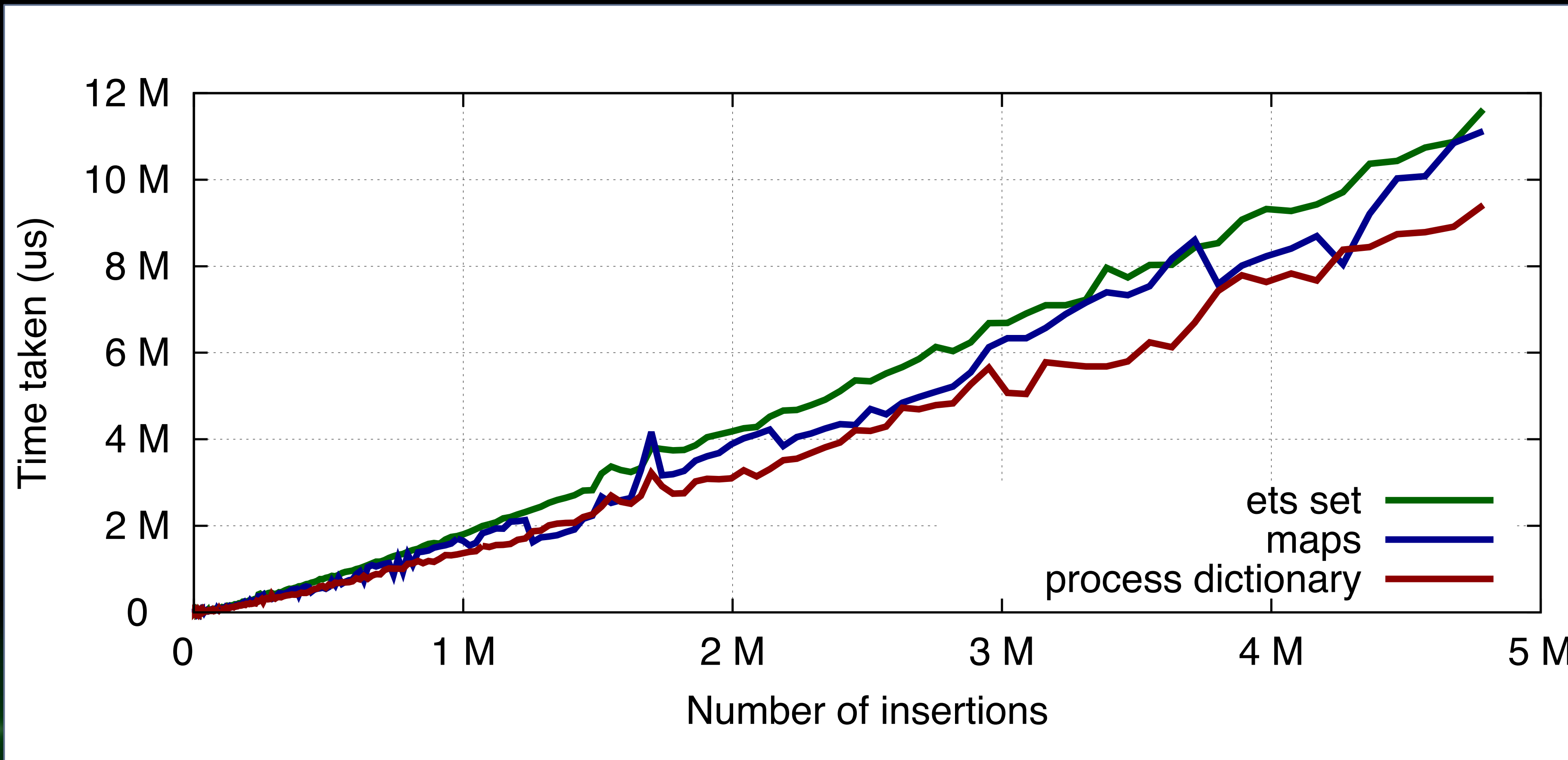
Lesson 1: Use Sharding

- Avoid having single process
- Use a pool of non-communicating shards
- `erlang:phash2(Key, NumberOfShards)`



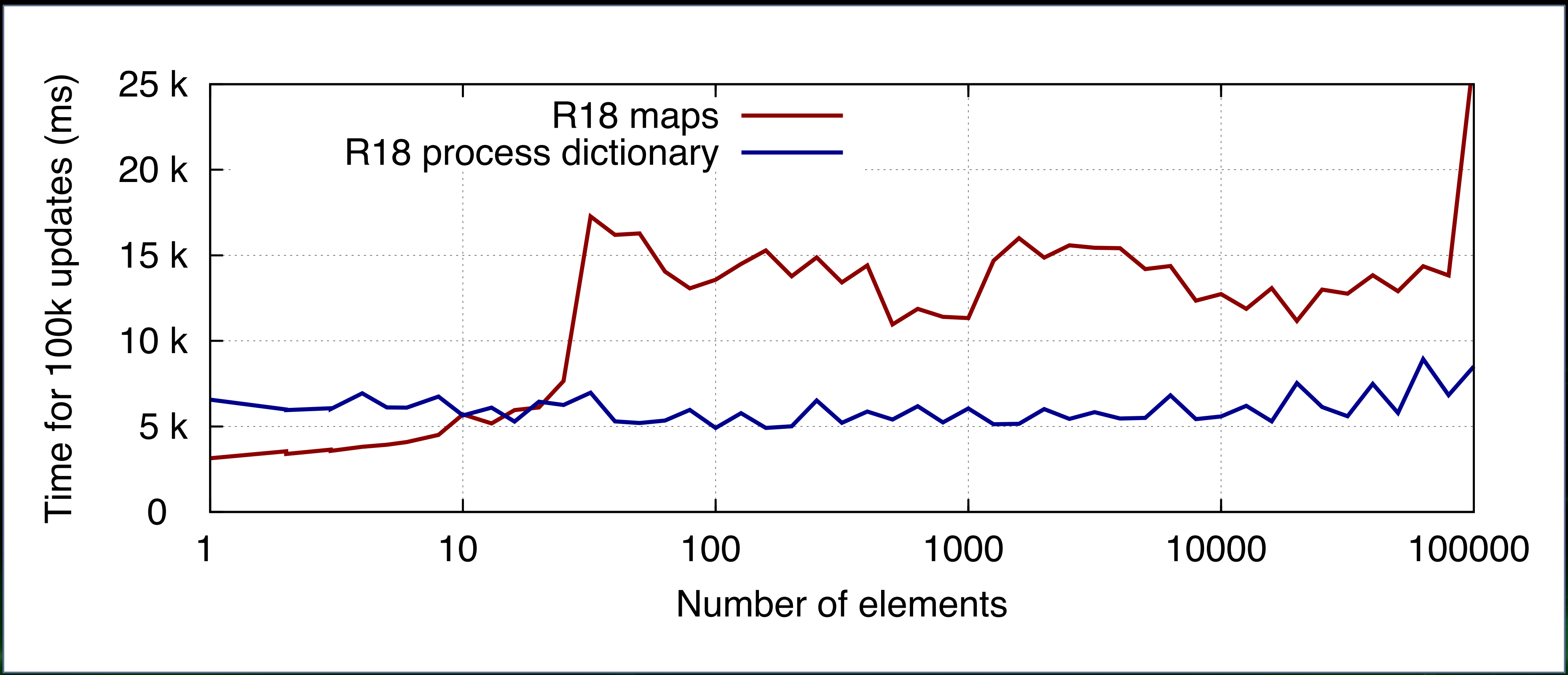
Lesson 2: Use Process Dictionary

- Quicker insertions in process dictionary compared to maps



Lesson 2: Use Process Dictionary

- Quicker update operations in process dictionary compared to maps (Erlang 18)



Lesson 3: Use Limited #Processes

- Don't create too many processes
  - pid2proc lock for process resolution

  - Scheduler has to manage the processes
    - Fixed number of schedulers

- Don't spawn/stop processes frequently
  - Process resolution again

Profiling Techniques

pstack: Stack Sampling

- pstack (gdb): sampling call stack over multiple threads
  - $ pstack 1234
  - $ gdb -p 1234
  - (gdb) thread apply all bt 25

```
Thread 5 (Thread 0x7efc4f2ba700 (LWP 14494)):
#0  0x00007efc61cbb199 in syscall () from /lib64/libc.so.6
#1  0x0000000000692533 in wait__ ()
#2  0x0000000000692622 in ethr_event_swait ()
#3  0x000000000068f4e2 in event_wait ()
#4  0x000000000068f82f in write_lock_wait ()
#5  0x00000000006908bf in rwmutex_normal_rwlock_wait ()
#6  0x000000000069158e in ethr_rwmutex_rwlock ()
#7  0x000000000055befd in erts_rwmtx_rwlock ()
#8  0x000000000055bfae in erts_smp_rwmtx_rwlock ()
#9  0x000000000055c6e8 in WLOCK_HASH ()
#10 0x00000000005638f5 in db_lookup_dbterm_hash ()
#11 0x00000000005406cb in do_update_counter ()
#12 0x0000000000540f89 in ets_update_counter_4 ()
#13 0x000000000043dc74 in process_main ()
#14 0x0000000000506cd5 in sched_thread_func ()
#15 0x00000000006917ea in thr_wrapper ()
#16 0x00007efc62179a51 in start_thread () from /lib64/libpthread.so.0
#17 0x00007efc61cbe93d in clone () from /lib64/libc.so.6
```

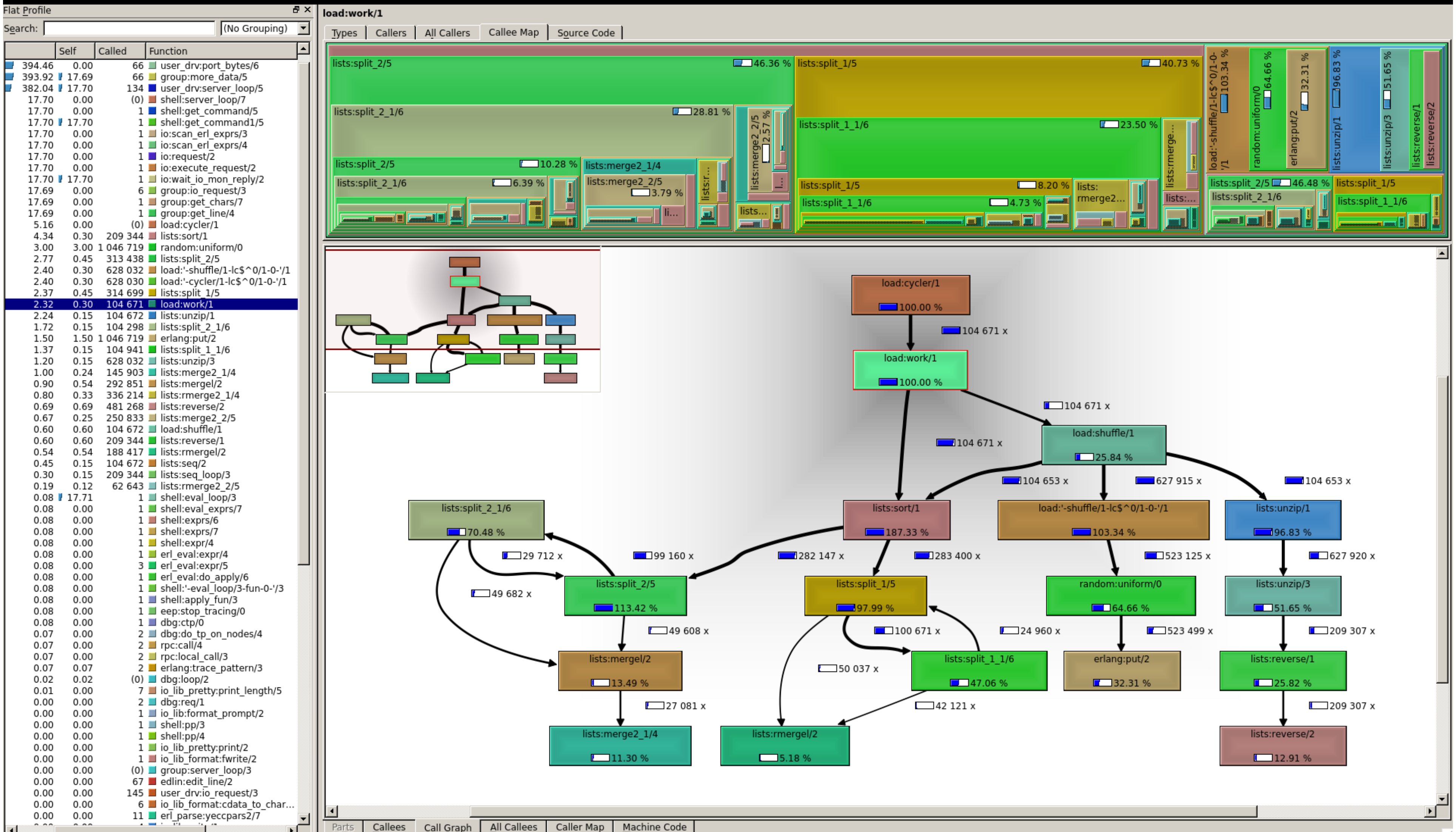
EEP: Erlang Easy Profiling

- https://github.com/virtan/eep
- Based on dbg module, low overhead trace ports
- Easy to use
- Kcachegrind visualization

Basic Usage

- Collect runtime data to local file
  - ```
    > eep:start_file_tracing("abc"),
      timer:sleep(10000),
      eep:stop_tracing().
    ```
- Convert to callgrind format
  - ```
    > eep:convert_tracing(“abc").
    ```
- Visualize with kcachegrind
  - ```
    $ kcachegrind callgrind.out.abc
    ```
    ```

EEP: Example Visualization



oprofile: System Level Profiling

- Initialization

```
$ opcontrol -deinit
$ echo 0 > /proc/sys/kernel/nmi_watchdog
```

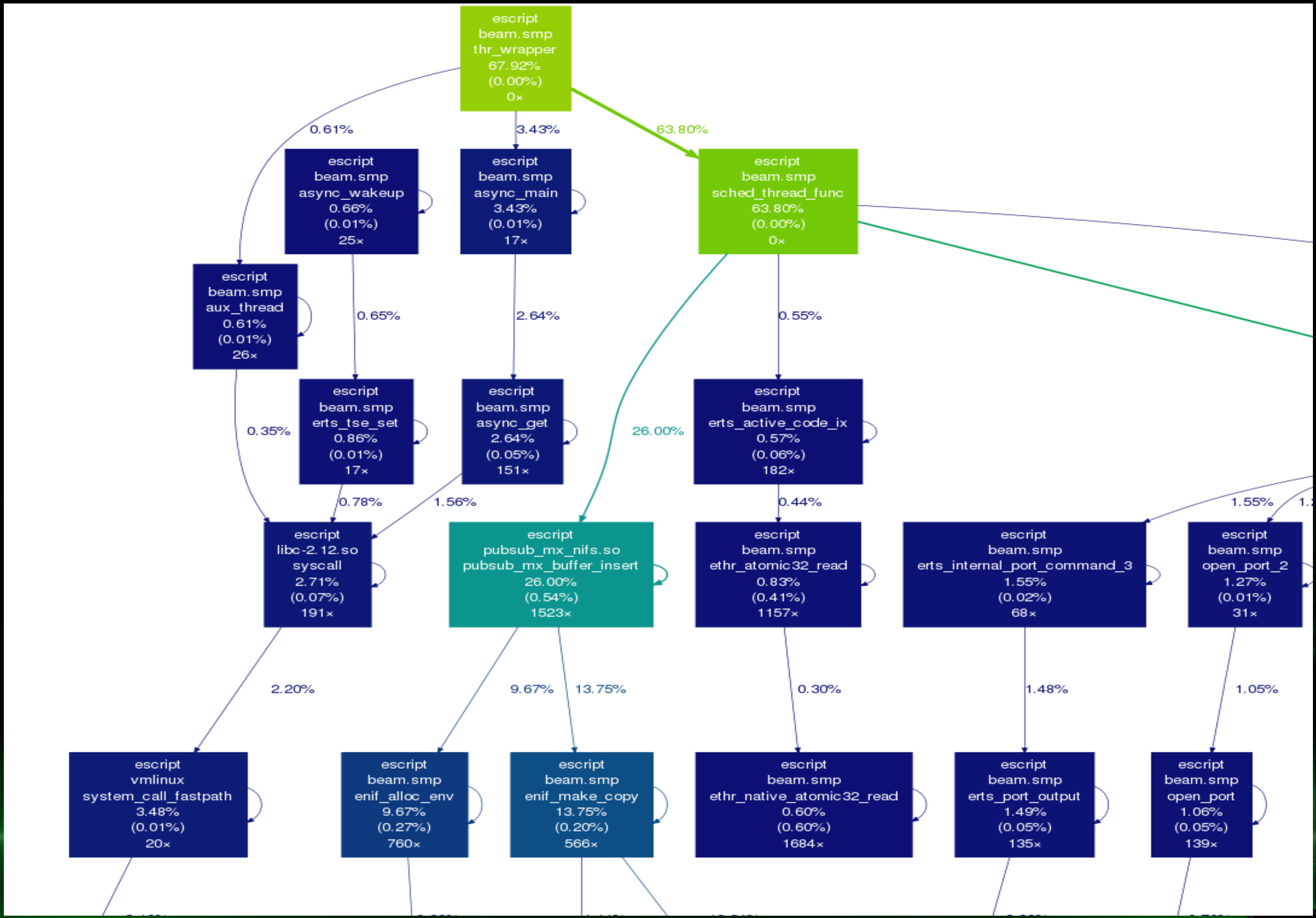
- Profiling

```
$ operf --vmlinux /usr/lib/debug/vmlinux --
callgraph ./rebar eunit apps=perf_test
```

- Reporting and Visualization

```
$ opreport -cgf | ./gprof2dot.py -f oprofile
--show-samples | dot -Tpng -o output.png
```

oprofile: Example



Case Study: Counters

Need for High Performance Counters

- Thousands of global counters
- Updated by several hundred processes every second

Libraries Used

- Exometer
- ETS
- Folsom

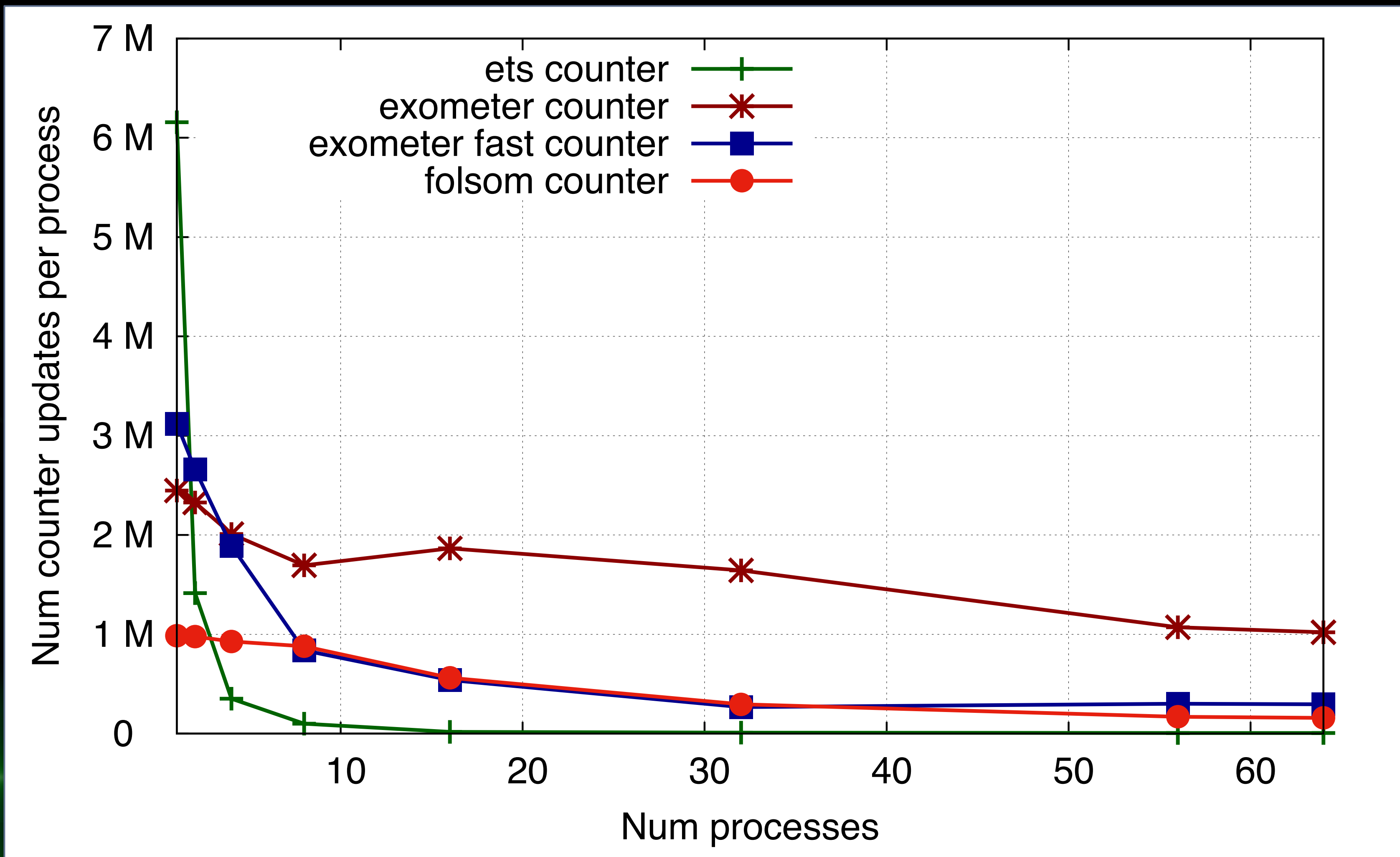
Benchmark Environment

- Platform
  - 2 CPU sockets
  - CPU - Intel(R) Xeon(R) CPU E5-2697 v3 @ 2.60GHz
  - 14 hardware threads per socket
  - 2 hyper-threaded threads for each hardware thread
- Erlang Runtime

```
Erlang/OTP 18 [erts-7.1] [source] [64-bit] [smp:56:56]
[async-threads:10] [kernel-poll:false]
```

Benchmark Results

- As number of processes increases, number of counter updates per process decreases



Limitations

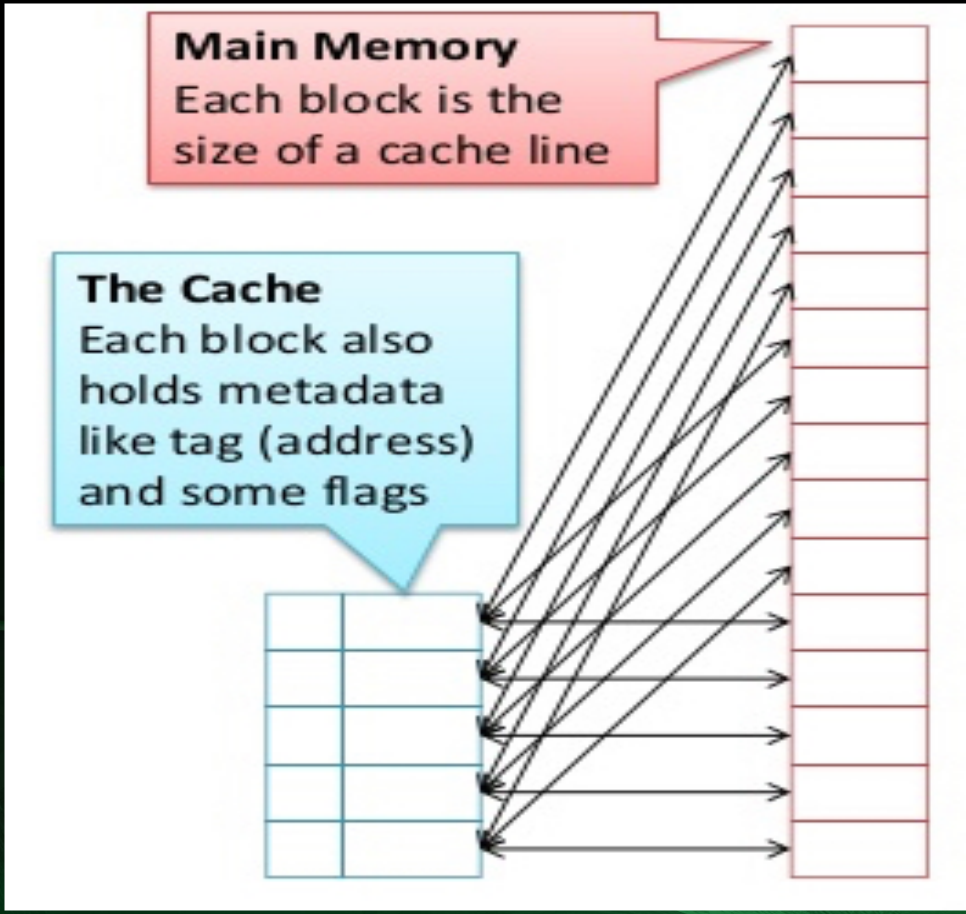
- Exometer:
  - counters – scales well but slow
  - fast counters – doesn't scale

- Lock contention in `ets:update_counter`()

Our Solution:

MZMETRICS

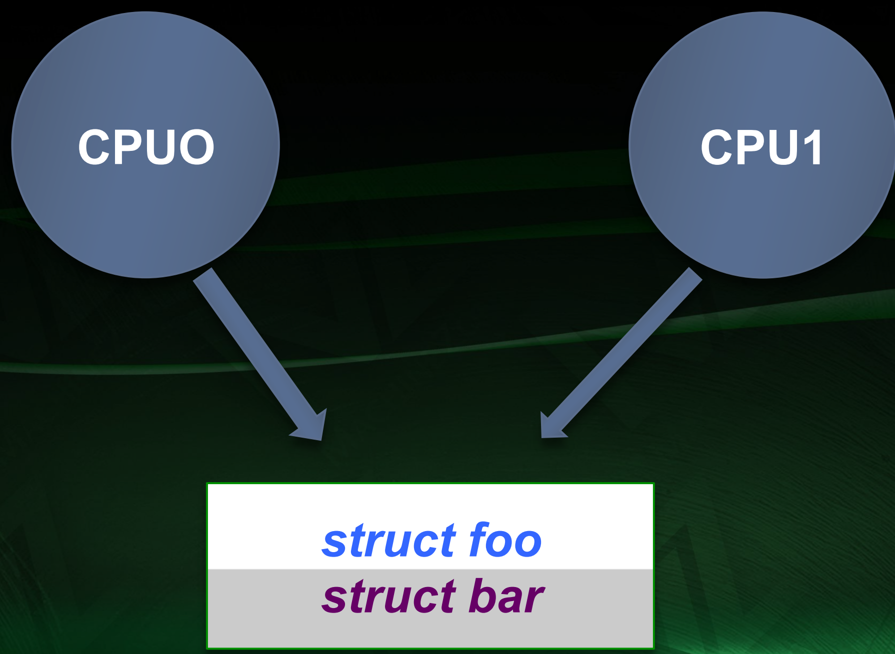
Refresher on Cacheline



Source: http://www.slideshare.net/yzt/modern-cpus-and-caches-a-starting-point-for-programmers/52

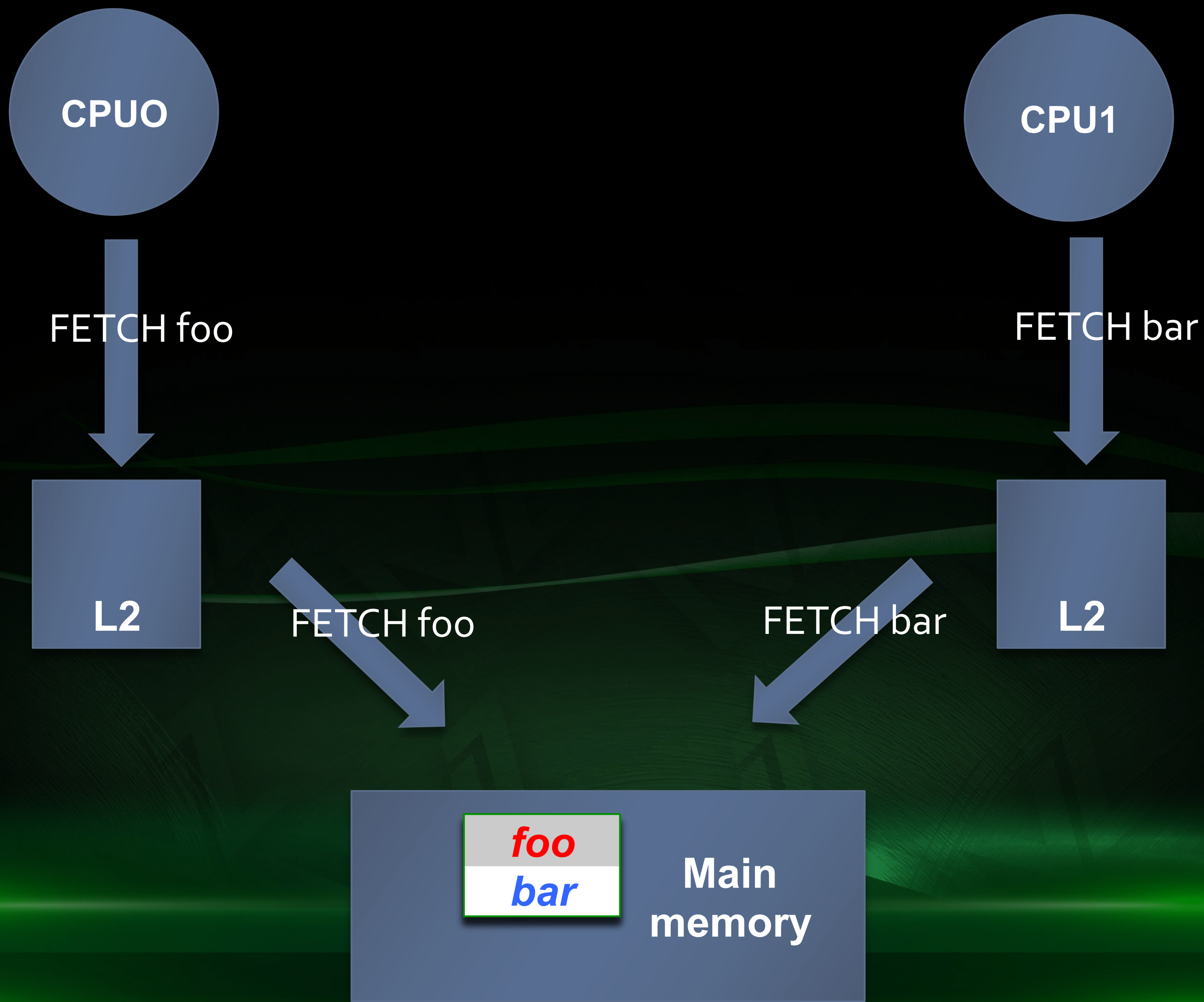
False Sharing (SMP)

Occurs when two CPUs access different data structures on the same cache line

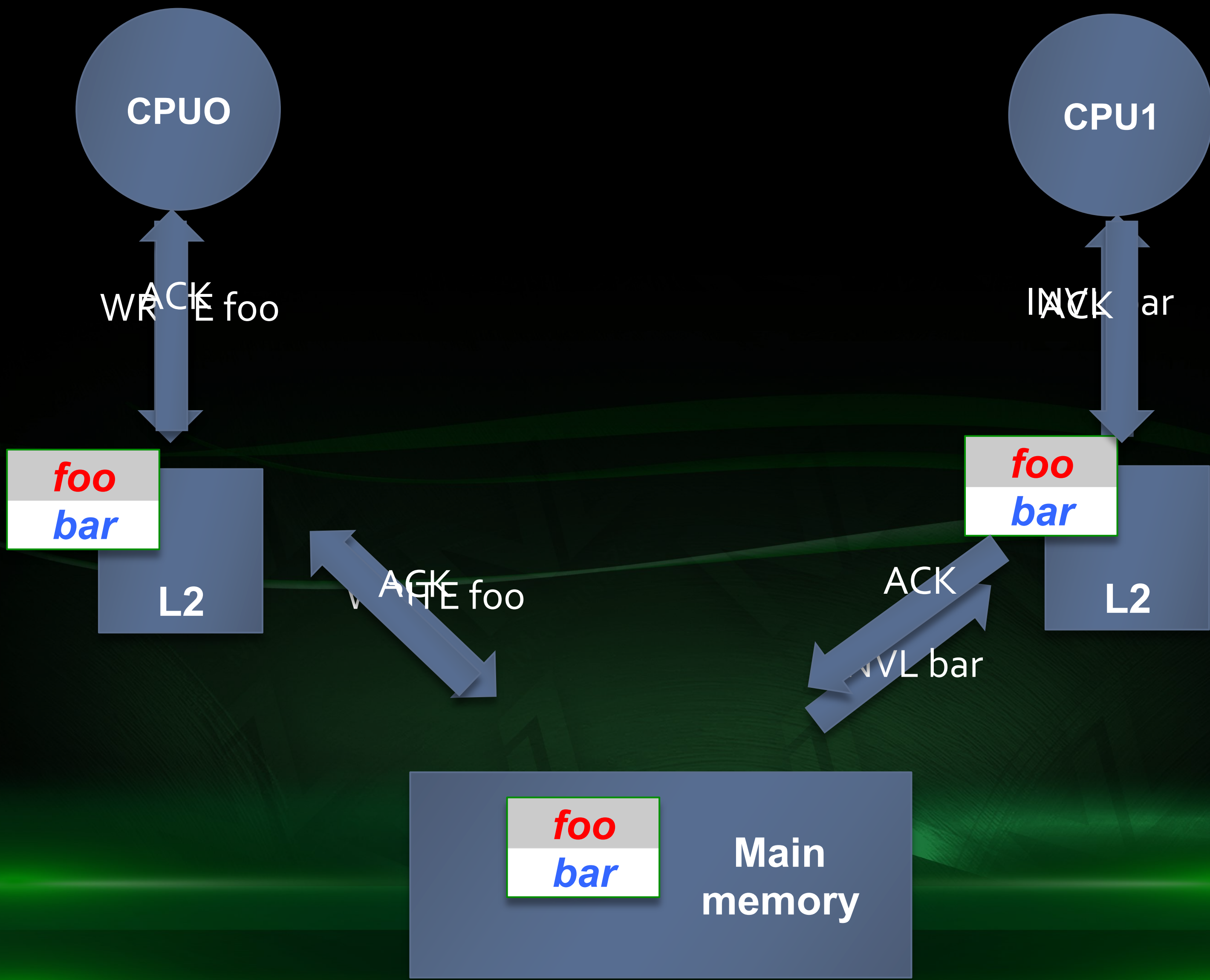


Cache line

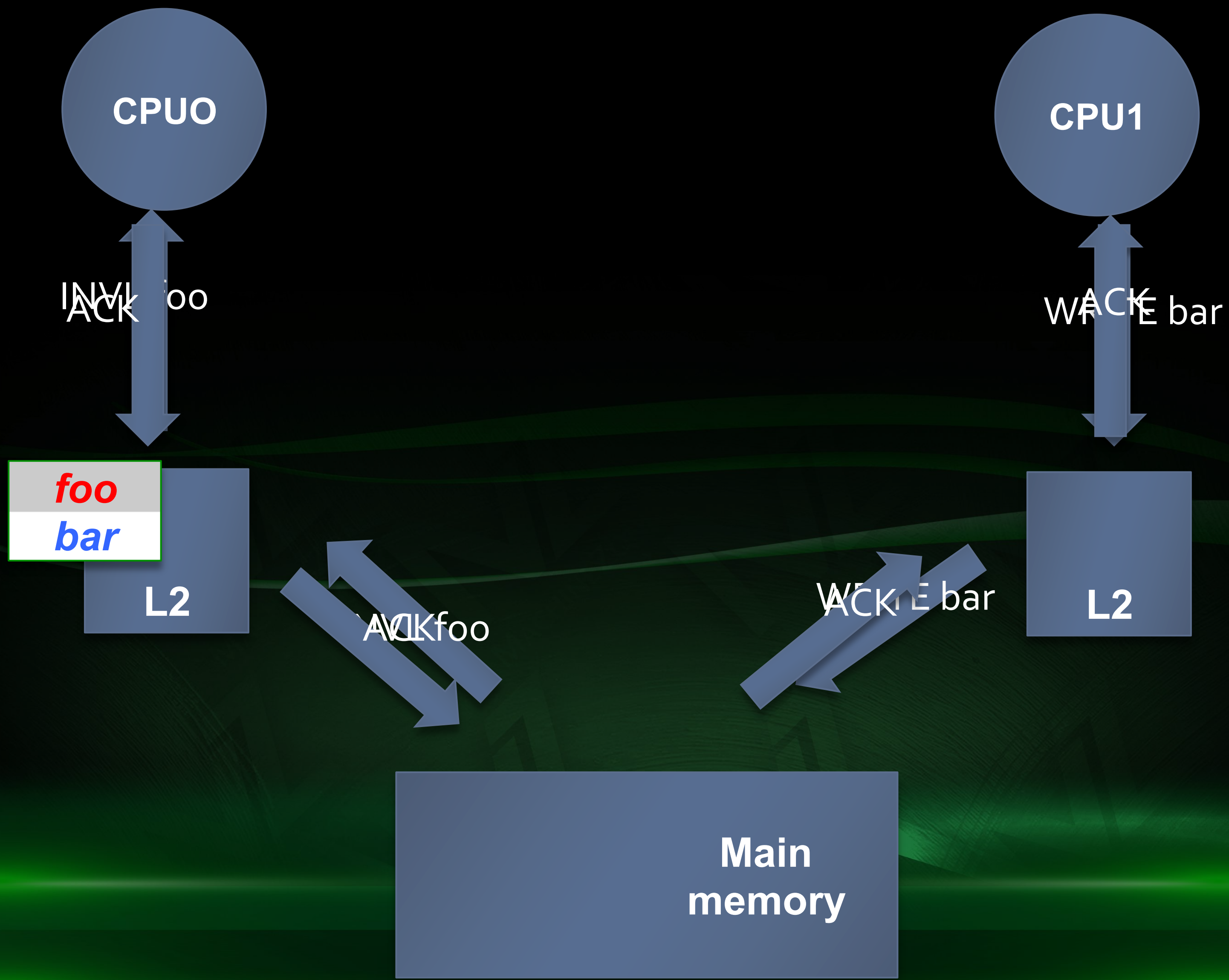
False Sharing (SMP)



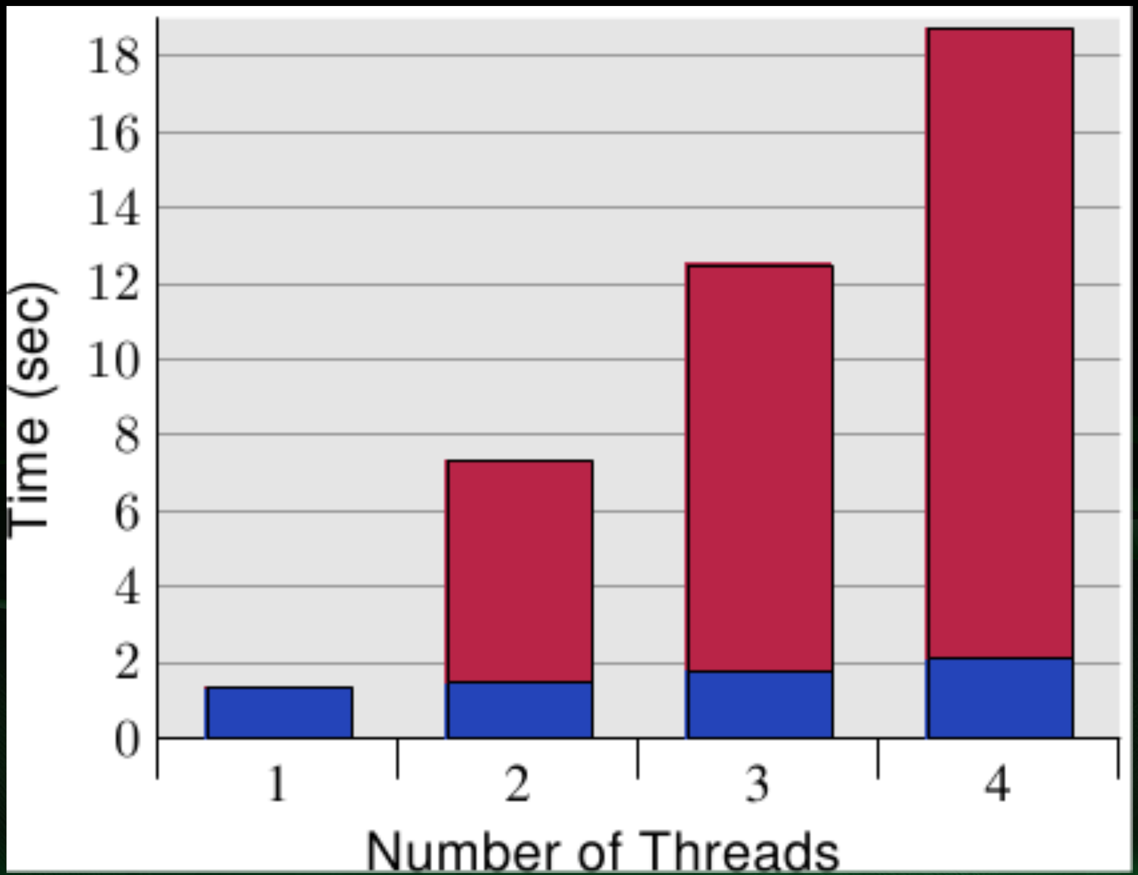
False Sharing (SMP)



False Sharing (SMP)



Benefits of Per-CPU Data



Time taken for 500 million counter increment operations on Intel Core 2 QX 6700

Source: http://lwn.net/Articles/256433/

Benefits of Per-CPU Data: Reasons

- Less overhead of cache coherency protocol
- Significantly higher benefits with QPI (Quick Path Interconnect)

Transactional Memory

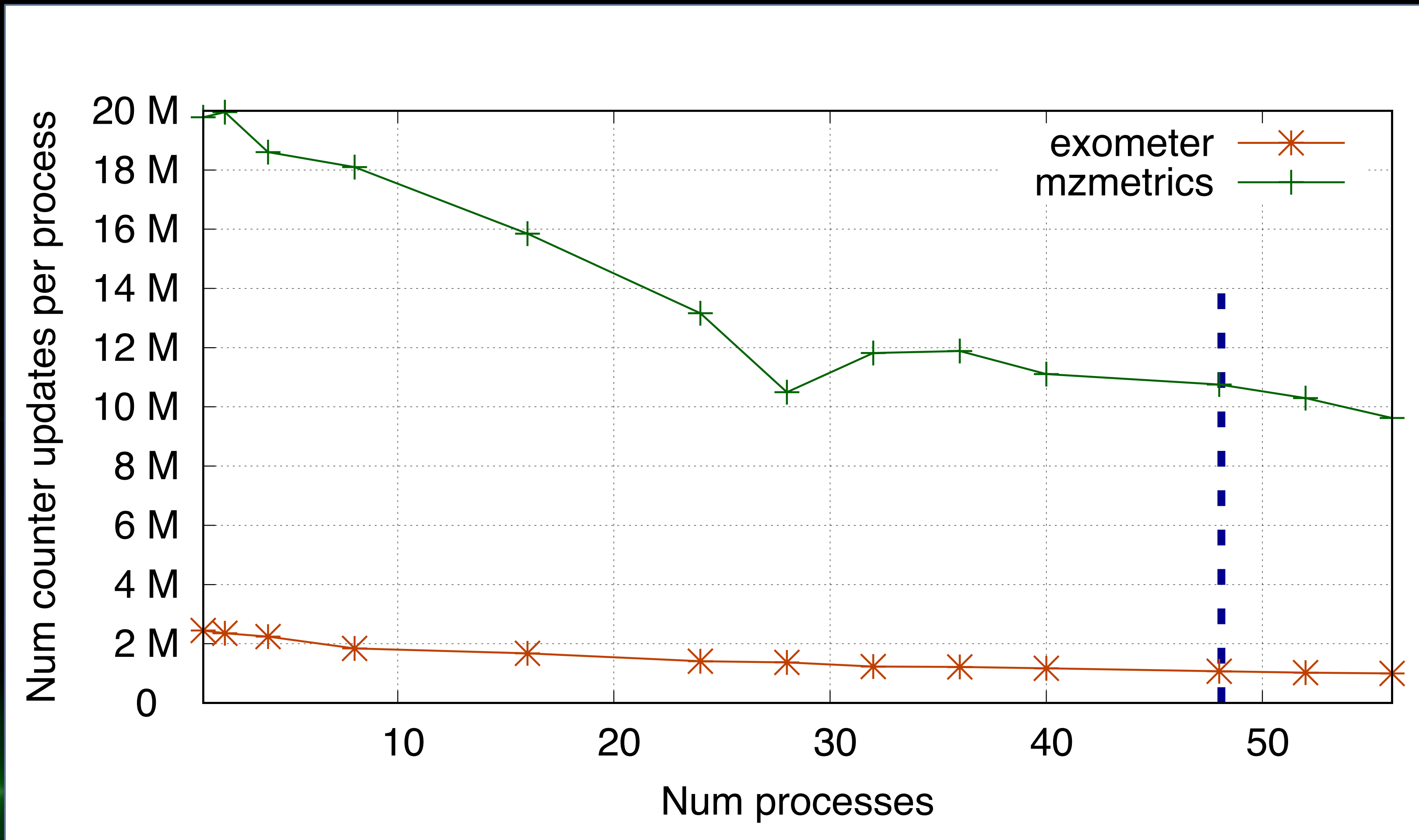
- Intel TSX (Transactional Synchronizations Extensions)
- Helps with locks in small critical sections
- Not yet available in x86 (disabled due to hardware bugs)

MZMETRICS

- Design based on per-CPU counters
- Counter Operations:
  - Create a new counter – requires lock
  - Read/update counter – no lock required
  - Delete counter – requires lock
- Implemented using NIF

MZMETRICS Evaluation

- 10X faster than Exometer counters



MZMETRICS Evaluation: Results

- 1 million counter updates with 48 processes
  - Exometer takes ~ 1 seconds
  - MZMETRICS takes ~0.1 seconds
- Sources:

  https://github.com/machinezone/mzmetrics

Takeaways

- Pitfalls
  - Message passing becomes expensive at scale
  - Large number of processes needing pid resolution from a global table causes contention
  - Existing libraries not amiable for fast counting
- Proposed Solutions
  - Use sharding & Limit message passing
  - Limit number of active processes
  - Use MZMETRICS for fast counters

We are Hiring

http://www.machinezone.com/careers