Better Mutual Exclusion on the Filesystem using Boost.AFIO

(asynchronous filesystem and file i/o)

Niall Douglas

Contents:

1. Quick overview of proposed Boost.AFIO, its motivation and its current status: v1 and v2
2. Overview of API design of Abstract Base Class `afio::algorithm::shared_fs_mutex::shared_fs_mutex`
3. High-level overview of our memory map lock algorithm
4. Detail of the memory map lock algorithm implementation
5. Benchmarks comparing our memory map lock algorithm implementation to lock files, byte ranges and atomic append locks on various filing systems

History and status of proposed Boost.AFIO (2012-)

Motivation

1. Why is it ***still*** so hard in 2016 to atomically update more than file at once?
   - Over usage of badly fitting SQLite
2. Why is it ***still*** so hard in 2016 to write DMA-friendly zero-copy storage code?
   - Ever more important with NVMe **4M** IOPS SSDs ...
3. Why is it ***still*** so hard in 2016 to write bug-free filesystem code?
   - The infamous directory tree deletion anti-pattern on Windows

Proposed Boost.AFIO v2 (> Oct 2015)

- “Bare metal” design
  - Exposes all the quirks of the host OS to user unfiltered
  - No threads, no resource nor memory management, no exceptions (we return lightweight mongrel-monadic *Outcomes* instead)
  - Performance never measurably worse than using host OS APIs directly - overhead often just *hundreds* of assembler opcodes
- Tight mapping between C++ type system and filing system object primitives
- Publicly exposes core file system template algorithms library into afio::algorithm::* (the AFIO “FTL”)
- Exemplar of simple, light, clean C++ 14/17 design

Proposed Boost.AFIO v2 - changes

Since ACCU 2016 (April):

- No AFIO v2 feature work apart from today's workshop
- BUT: Beginnings of a test suite based on CTest, Boost.KernelTest and threadsafe CATCH
    - KernelTest lets you write unit test suites *functionally* instead of imperatively
        - Parameter permutation lists, preconditions, postconditions
    - Maximum use of CPU cores, bias towards CI driven soak testing
    - In Progress: Automatic CI ASan, MSan, TSan, UBSan, valgrind, libfuzzer
    - Soon: Automatic *edge coverage* and *code bloat* calculation
    - Despite immaturity, **lots of AFIO v2 bugs fixed already**

Proposed Boost.AFIO v2 - changes

- Huge improvements to Boost-lite
  - AFIO now has a cmake3 based build system yay!
    - As does Outcome, KernelTest and Boost-lite itself
  - Via Boost-lite build **all Boost-lite libraries have**:
    - Automatic CMake, CTest, CDash and (soon) CPack
    - Automatic binary and source uploads to CDash
    - Automatic Travis and Appveyor CI
    - Automatic use of C++ Modularisation & precompiled headers
    - Automatic submodularisation & dependency tracking
    - Automatic flat hierarchy or "single download and drop in" header-only installation
    - Automatic ABI versioning and hard version binds
  - Boost-lite remains *very* immature though!

Login All Dashboards Plans & Pricing Support Wednesday, September 07 2016 06:13:10 E

Boost.AFIO

Dashboard Calendar Previous Current Project

No file changed as of **Wednesday, September 07 2016 - 01:00 EDT**

3 hours ago: 10 tests not run on Linux-3.13.0-40-generic-x86_64

3 hours ago: 12 errors introduced on Linux-3.13.0-40-generic-x86_64

17 hours ago: 10 tests not run on Linux-3.13.0-40-generic-x86_64

17 hours ago: 12 errors introduced on Linux-3.13.0-40-generic-x86_64

1 days ago: 1 test failed on Windows-AMD64

See full feed

Experimental

Site	Build Name	Update	Configure		Build		Test			
		Files	Error	Warn	Error	Warn	Not Run	Fail	Pass	Build Time
TravisLinuxWorker	Linux-3.13.0-40-generic-x86_64		0	0	12	0	10	0	0	2 hours ago
TravisDocumentation	Linux-3.13.0-40-generic-x86_64		0	0	0	0				2 hours ago
APPVYR-WIN	Windows-AMD64		0	6	0	0	0	0	30	3 hours ago

CDashPro 2.3.0 © Kitware | Report problems | Privacy Policy | 0.01s

boost.afio-v2.0-binaries-win64-201608262229.zip	26-Aug-2016 22:29	5377132
boost.afio-v2.0-binaries-win64-201608270003.zip	27-Aug-2016 00:03	5377146
boost.afio-v2.0-binaries-win64-201608292307.zip	29-Aug-2016 23:07	5338070
boost.afio-v2.0-binaries-win64-201609030006.zip	03-Sep-2016 00:06	5338069
boost.afio-v2.0-binaries-win64-201609061654.zip	06-Sep-2016 16:54	5493508
boost.afio-v2.0-binaries-win64-201609070715.zip	07-Sep-2016 07:15	5494875
boost.afio-v2.0-source-201608131743.tar.xz	13-Aug-2016 17:43	7066248
boost.afio-v2.0-source-201608131810.tar.xz	13-Aug-2016 18:11	7066340
boost.afio-v2.0-source-201608131829.tar.xz	13-Aug-2016 18:30	7068964
boost.afio-v2.0-source-201608131935.tar.xz	13-Aug-2016 19:36	7069012
boost.afio-v2.0-source-201608141334.tar.xz	14-Aug-2016 13:35	7069032
boost.afio-v2.0-source-201608141813.tar.xz	14-Aug-2016 18:13	7081544
boost.afio-v2.0-source-201608171941.tar.xz	17-Aug-2016 19:41	7131376
boost.afio-v2.0-source-201608171943.tar.xz	17-Aug-2016 19:44	7131248
boost.afio-v2.0-source-201608171950.tar.xz	17-Aug-2016 19:50	7131292
boost.afio-v2.0-source-201608181945.tar.xz	18-Aug-2016 19:45	7130588
boost.afio-v2.0-source-201608192001.tar.xz	19-Aug-2016 20:02	7131556
boost.afio-v2.0-source-201608212029.tar.xz	21-Aug-2016 20:29	7153236
boost.afio-v2.0-source-201608222020.tar.xz	22-Aug-2016 20:20	7154696
boost.afio-v2.0-source-201608230917.tar.xz	23-Aug-2016 09:17	7154928
boost.afio-v2.0-source-201608232041.tar.xz	23-Aug-2016 20:41	7156384
boost.afio-v2.0-source-201608242239.tar.xz	24-Aug-2016 22:41	7159840
boost.afio-v2.0-source-201608262006.tar.xz	26-Aug-2016 20:06	7160264
boost.afio-v2.0-source-201608262054.tar.xz	26-Aug-2016 20:55	7160752
boost.afio-v2.0-source-201608262229.tar.xz	26-Aug-2016 22:29	7160276
boost.afio-v2.0-source-201609041922.tar.xz	04-Sep-2016 19:22	7171832
boost.afio-v2.0-source-201609041929.tar.xz	04-Sep-2016 19:30	7171916
boost.afio-v2.0-source-201609050824.tar.xz	05-Sep-2016 08:24	7171984
boost.afio-v2.0-source-201609060928.tar.xz	06-Sep-2016 09:28	7202656
boost.afio-v2.0-source-201609061652.tar.xz	06-Sep-2016 16:52	7203640
boost.afio-v2.0-source-201609070713.tar.xz	07-Sep-2016 07:13	7206252
boost.outcome-v1.0-source-201608131653.tar.xz	13-Aug-2016 16:54	614640
boost.outcome-v1.0-source-201608131738.tar.xz	13-Aug-2016 17:39	614600
boost.outcome-v1.0-source-201608131807.tar.xz	13-Aug-2016 18:08	614808
boost.outcome-v1.0-source-201608131827.tar.xz	13-Aug-2016 18:28	614820
boost.outcome-v1.0-source-201608141807.tar.xz	14-Aug-2016 18:08	615232
boost.outcome-v1.0-source-201608171939.tar.xz	17-Aug-2016 19:40	617892
boost.outcome-v1.0-source-201608181942.tar.xz	18-Aug-2016 19:43	617920
boost.outcome-v1.0-source-201608191957.tar.xz	19-Aug-2016 19:58	617876
boost.outcome-v1.0-source-201608212027.tar.xz	21-Aug-2016 20:28	623960

Top level AFIO v2 afio::algorithm::shared_fs_mutex::* API overview

Prereq: Boost.Outcome (mongel monads)

- Policy driven underlying implementation basic_monad<T, EC, E> is aliased into convenience typedefs:
  - outcome<T> = empty OR T OR std::error_code OR std::exception_ptr
  - result<T> = empty OR T OR std::error_code
  - option<T> = empty OR T
- Has identical API to a std::future<T> or std::optional<T> (e.g. get(), value(), has_value() etc)

Prereq: AFIO v2 byte range lock API

```
class io_handle : public handle {
    using extent_type = unsigned long long;
    class extent_guard;  // RAII guard
    virtual result<extent_guard> lock(extent_type offset,
extent_type bytes, bool exclusive = true, deadline d =
deadline()) noexcept;
};
```

Passes straight through to syscall, insane semantics and all if your POSIX OS is mad

afio::algorithm::shared_fs_mutex

There are these implementations of shared_fs_mutex in AFIO v2:

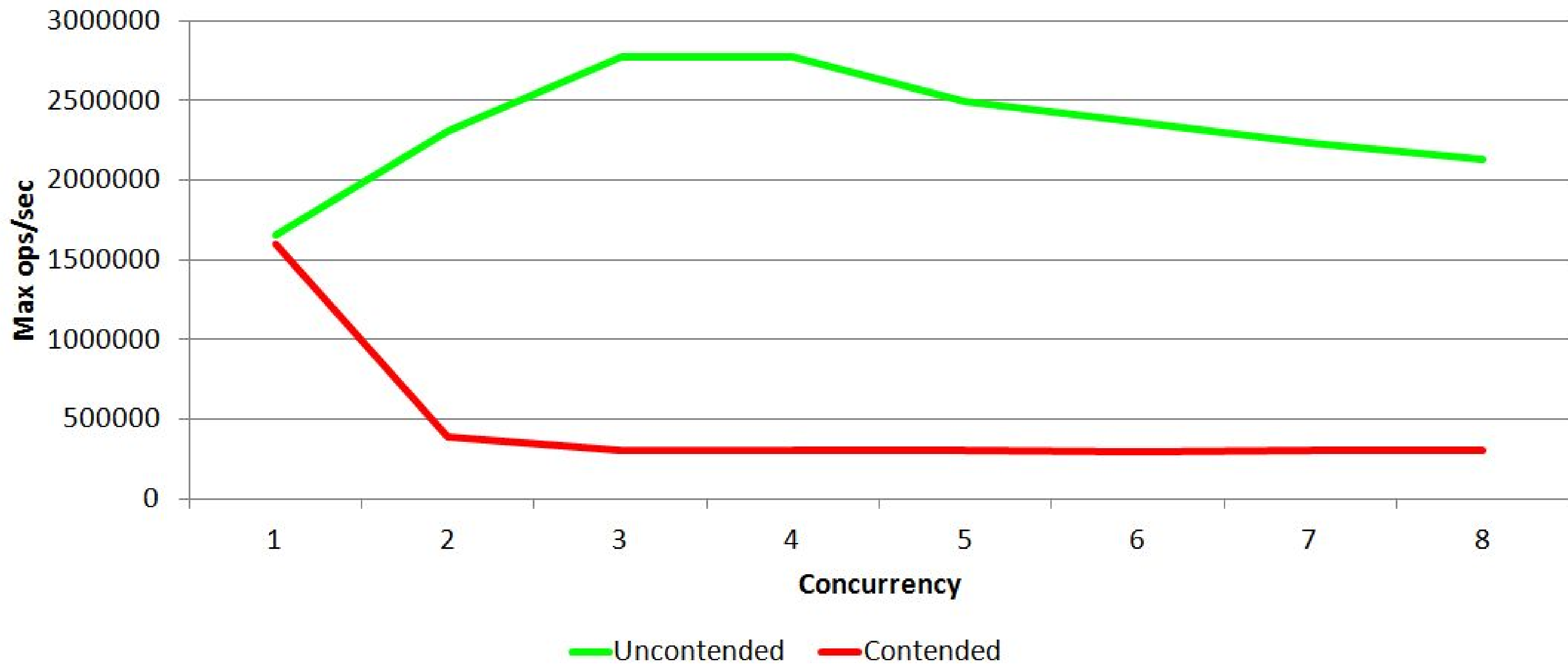
1. shared_fs_mutex::lock_files
2. shared_fs_mutex::byte_ranges
3. shared_fs_mutex::atomic_append (ACCU talk)
4. shared_fs_mutex::memory_map (today)

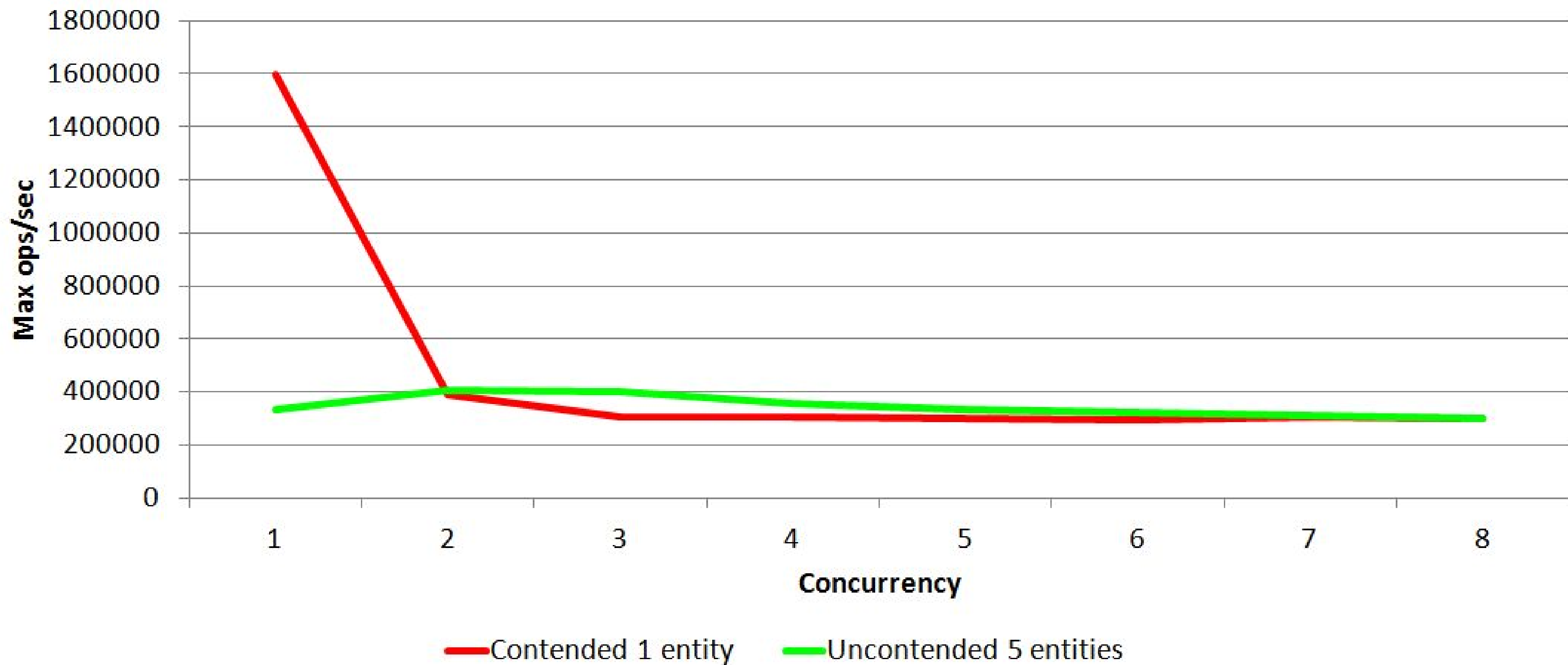
All implement the abstract base class shared_fs_mutex::shared_fs_mutex

shared_fs_mutex abstract base class

```
class shared_fs_mutex {
      struct entity_type { value : 63; exclusive: 1; };
      using entities_type = gsl::span<entity_type>;
      class entities_guard;  // RAII guard
      (virtual) result<entities_guard> lock(entities_type
entities, deadline d = deadline(), bool spin_not_sleep =
false) noexcept;
      (virtual) result<entities_guard> try_lock(entities_type
entities) noexcept { return lock(std::move(entities),
deadline(std::chrono::seconds(0))); }
};
```

Why does the shared FS mutex API lock many entities?


Scaling of byte range locks to concurrency on NTFS
(4 core 8 thread CPU)
Max ops/sec
3000000
2500000
2000000
1500000
1000000
500000
0
1
2
3
4
5
6
7
8
Concurrency
Uncontended
Contended

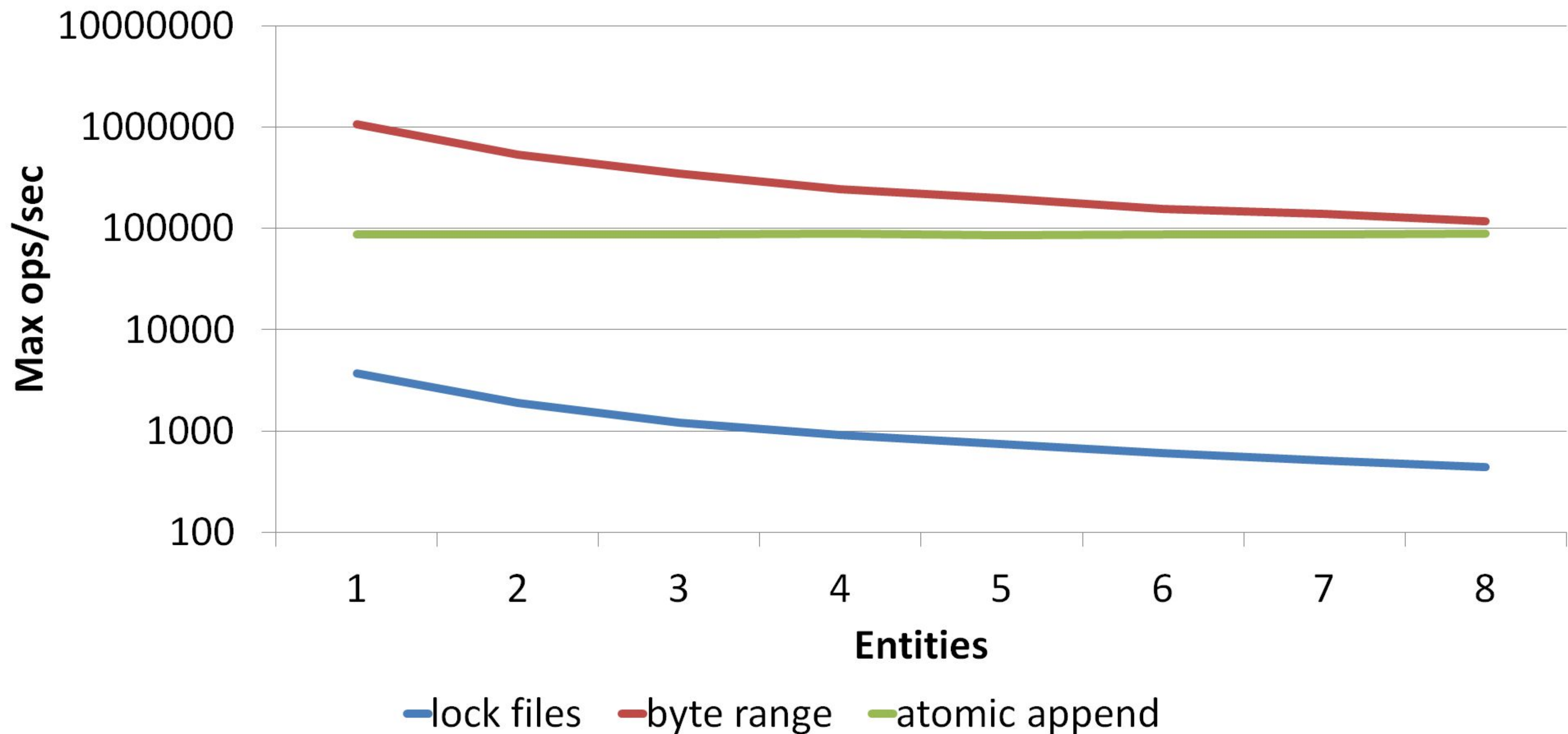

How many uncontended entities equals single contended byte range lock
(4 core 8 thread CPU)
Max ops/sec
1800000
1600000
1400000
1200000
1000000
800000
600000
400000
200000
0
1
2
3
4
5
6
7
8
Concurrency
Contended 1 entity
Uncontended 5 entities

afio::algorithm::shared_fs_mutex

- How entities are mapped onto the file system depends on the implementation, so:
  - shared_fs_mutex::lock_file maps entities into 16 character hexadecimal files opened with O_EXCL
  - shared_fs_mutex::byte_ranges maps entities into single byte offsets into the shared lock file
    - Contention is handled by backing off all preceding locks and randomising the list before trying again, starting with the entity which was contended last try. This allows sleeping until contended lock becomes free

afio::algorithm::shared_fs_mutex

- shared_fs_mutex::atomic_append (see my ACCU 2016 talk) solves the problem of *inverse log scaling* to entity count in lock_files and byte_ranges
  - atomic_append pretty much always wins once you are locking >= 15 entities uncontended
  - Has easily best worst case performance in most heavy concurrency use cases, avoiding the "scalability hole" which most kernel filesystem lock implementations fall into after a certain load of entities and concurrency


Scaling of four concurrent users of the lock algorithms to entities on NTFS
10000000
1000000
100000
10000
1000
100
Max ops/sec
1
2
3
4
5
6
7
8
Entities
lock files
byte range
atomic append

afio::algorithm::shared_fs_mutex

Last remaining class of shared FS mutex is a very high performance one based entirely in userspace using shared memory atomics

- Comes with *lots* of caveats … more on that shortly …
- But it is probably likely most folk will default to it due to its sheer raw performance which is 10x-20x more than any other shared FS mutex in AFIO

(I have noticed people tend to default to best average performance rather than best worst case performance)

Further reading:

AFIO v2 API reference:

https://ned14.github.io/boost.afio

Abstract base class

algorithm::shared_fs_mutex API reference:

http://goo.gl/s23ecD

ACCU 2016 atomic append talk video:

https://www.youtube.com/watch?v=elegewDwm64

The memory map lock algorithm - Overview

Questions before we begin this section?

The shared memory map algorithm

- Only viable if there are no networked drive users
  - Therefore will also need a *fallback lock* which IS compatible with networked drive users
  - And the ability for all users of the lock to jointly degrade to that fallback lock in safe fashion if a networked drive user turns up
    - Which in turns means we need some way of portably detecting the arrival of a networked drive user

The memory_map algorithm

To construct the shared mutex instance:

1. If the lockfile doesn't exist, ***race free*** create a mapfile in **/tmp** and write its path into the lockfile. Otherwise open the lockfile and read the path of the mapfile somewhere in **/tmp**. Try opening that, if we can't then **degrade the lock**
2. Map the lockfile into memory **read-only** and the **/tmp** mapfile into memory read-write

(Big assumption here is that whatever is said to be **/tmp** by the OS is visible to all local processes, but not to any networked drive users)

The memory_map algorithm

To lock:

1. Has the shared memory map been degraded by a networked drive user? If so, pass through lock request to fallback lock
2. Hash each entity and modulus by total map entries
3. Try locking the spin lock at each entity’s hash index. If fail to lock, unlock everything, randomise and retry

To unlock:

1. Unlock all the spin locks locked earlier (or fallback lock)

The memory_map algorithm

Characteristics:

- Does at least one atomic operation per entity
    - But cascades atomic ops if any contended, and SMP and especially NUMA have a very finite total system atomic operation bandwidth
- No ability to sleep the CPU
    - Spin locks do have a counter and do call sleep(1ms) eventually
- Quality of hash function very important
    - Collision probing left and right substantially increases cache line bouncing, whole system performance damage is very severe
- This algorithm is fundamentally *anti-social* to all other code on a machine … but it’s fast, very fast

The memory map lock algorithm - Detail

Questions before we begin this section?

memory_map<...>

```
template <
  template <class> class Hasher = fnv1a_hash,
  size_t HashIndexSize = 4096,
  class SpinlockType = shared_spinlock<>
> class memory_map : public shared_fs_mutex
```

- fnv1a_hash and shared_spinlock<Policy> come from Boost-lite https://github.com/ned14/boost-lite
- 1Mb + 0 = lock in use, 1Mb + 1 = map in use

memory_map<>::lock()

1. Is lock degraded (lockfile[0] is zero)? If so:
   a. Have we only just degraded? If so:
      i. Release shared lock map-in-use byte 1Mb + 1
      ii. Exclusive lock map-in-use byte 1Mb + 1 and immediately release
   b. Redirect to fallback lock
2. Hash and modulus to mapfile entries every entity in the lock list in an auto-vectorising SIMD friendly fashion, eliding index collisions (with exclusive bit upgrade)

memory_map<>::lock()

3. For every entity index:
    a. If exclusively locking, try to exclusive lock the spinlock at that index, else try to shared lock it
    b. If lock fails:
        i. Unlock everything locked so far
        ii. Check if deadline has passed, if so return ETIMEDOUT
        iii. Swap contended index with first index
        iv. ~~std::random_shuffle()~~ rest of index list
        v. Yield thread context and retry sequence

Questions?

memory_map<>::unlock()

1. If lock degraded, pass to fallback lock
2. Hash and modulus to mapfile entries every entity in the lock list in an auto-vectorising SIMD friendly fashion, eliding index collisions (with exclusive bit upgrade)
3. For every entity index:
   a. If exclusively unlocking, try to exclusive unlock the spinlock at that index, else try to shared unlock it
4. Is lock newly degraded (lockfile[0] is zero)? If so, release shared lock map-in-use byte 1Mb +1

Questions?

memory_map<>::fs_mutex_map()

Remember:

- 1Mb + 0 = “lock in use” byte range lock offset
  - Used to detect other users of the *lock*
- 1Mb + 1 = “map in use” byte range lock offset
  - Used to detect other users of the *map*

1. Open the lockfile, creating if needed, caching only reads (not writes)
2. Try to exclusive lock in-use byte at 1Mb + 0

memory_map<>::fs_mutex_map()

3. If success (i.e. lock is NOT in use):
   a. Truncate lockfile to zero
   b. Create a randomly named mapfile in /tmp
   c. Truncate it to HashIndexSize
   d. Write path of /tmp mapfile into lockfile
   e. Shared lock map-in-use byte at 1Mb + 1
   f. Convert exclusive lock in-use byte at 1Mb + 0 into a shared lock
   g. Map mapfile (rw) and lockfile (ro) into memory

memory_map<>::fs_mutex_map()

3. If failure (i.e. lock IS in use):
   a. Shared lock in-use byte at 1Mb + 0 (blocks)
   b. Read path from lockfile and try to open
   c. If not found, write zeroes all over lockfile if needed and return EBUSY
   d. If found, shared lock map-in-use byte at 1Mb + 1
   e. Map mapfile (rw) and lockfile (ro) into memory

Questions?

memory_map<>::~memory_map()

1. Unlock in-use byte at 1Mb + 0 and map-in-use byte at 1Mb + 1
2. Exclusive lock in-use byte at 1Mb + 0. If success:
   a. Write 4Kb of zeroes to lockfile
   b. Truncate lockfile to zero
   c. Unlink /tmp mapfile
   d. Release all locks, lock is reset

Questions?

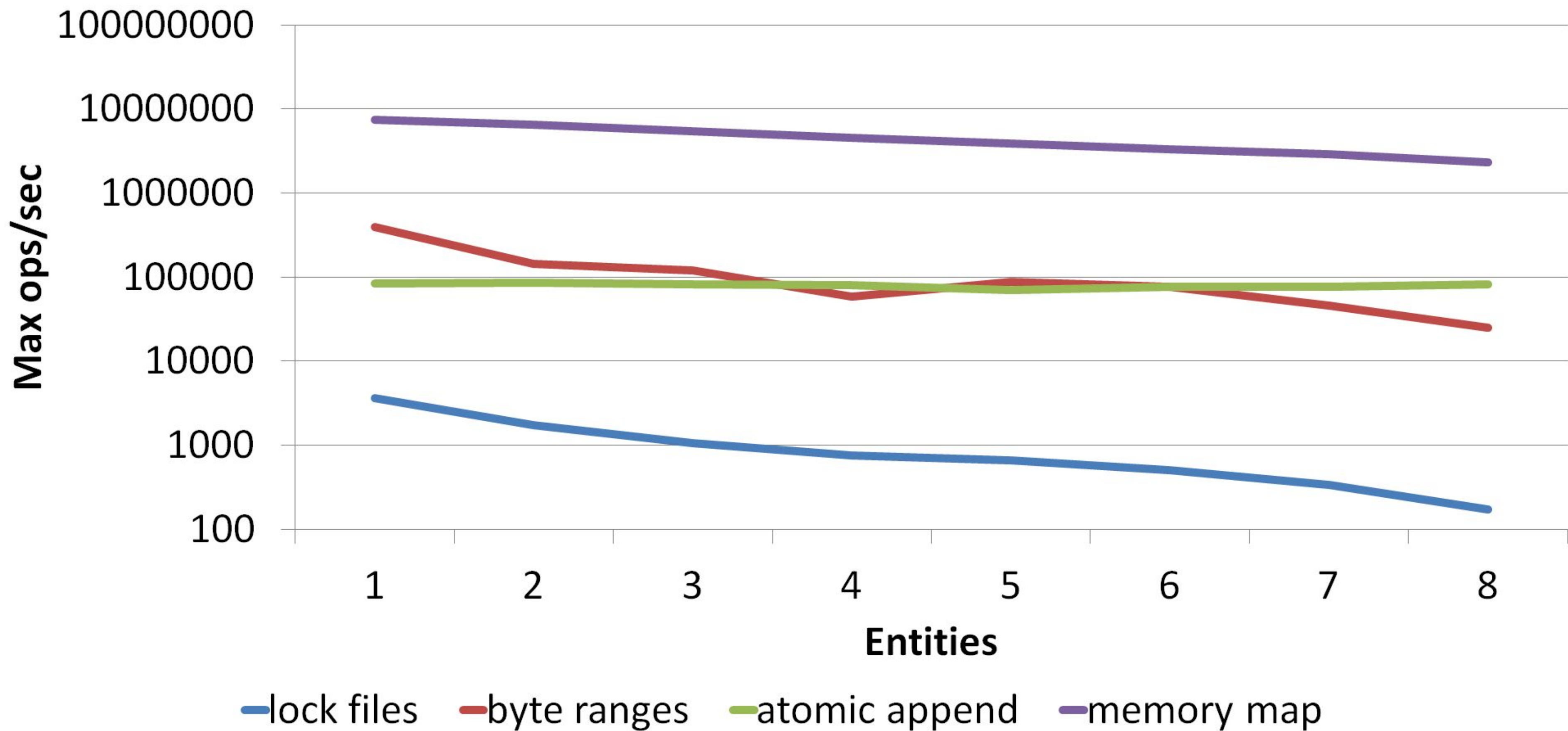
Benchmarks!

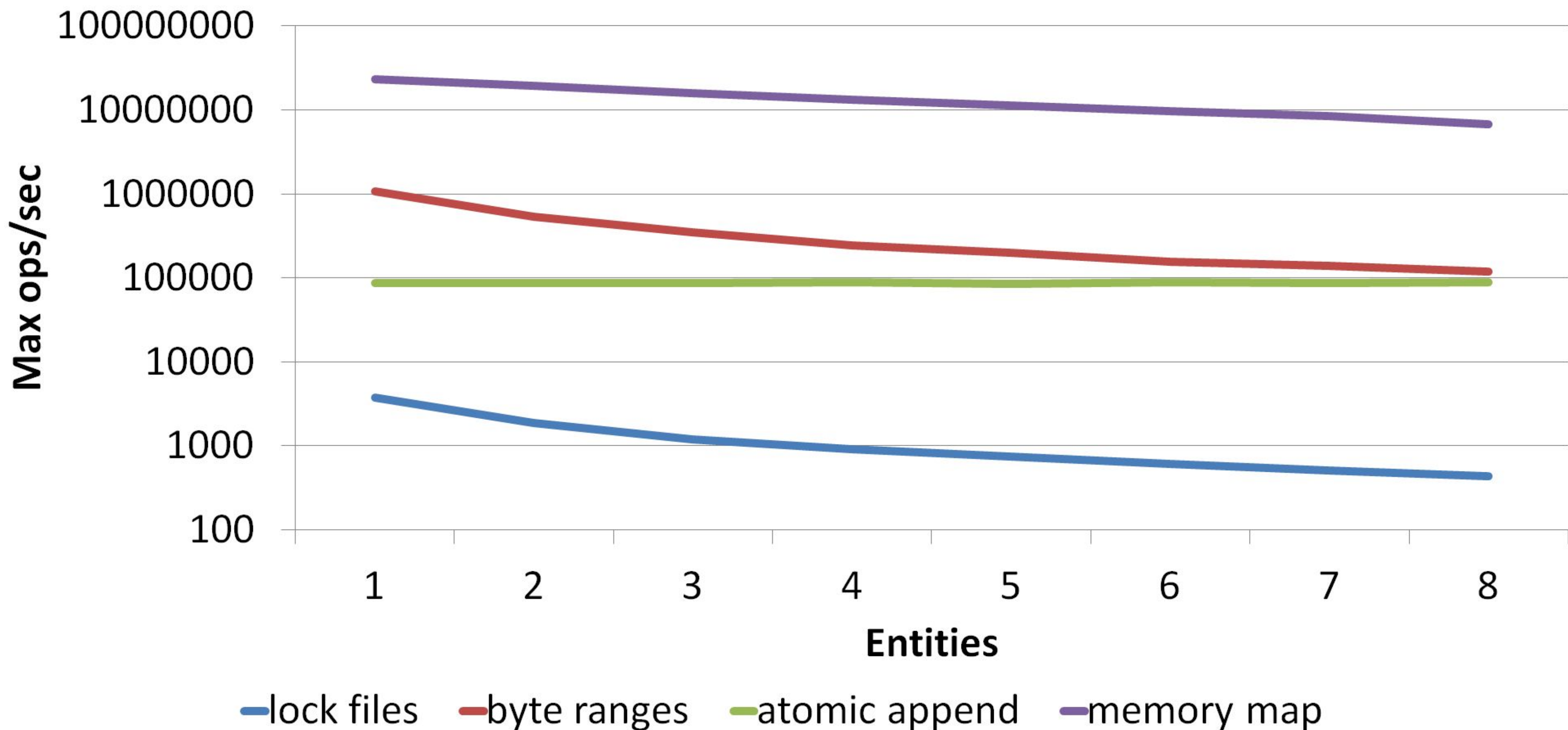
The following benchmarks are for:

2 core 4 thread 4.125Gb/sec main memory bandwidth
GenuineIntel Intel(R) Core(TM) i5 CPU M 540 @ 2.53GHz (2008 era laptop)
Microsoft Windows 10.0.14393.103

Known deficiencies in this memory_map implementation:

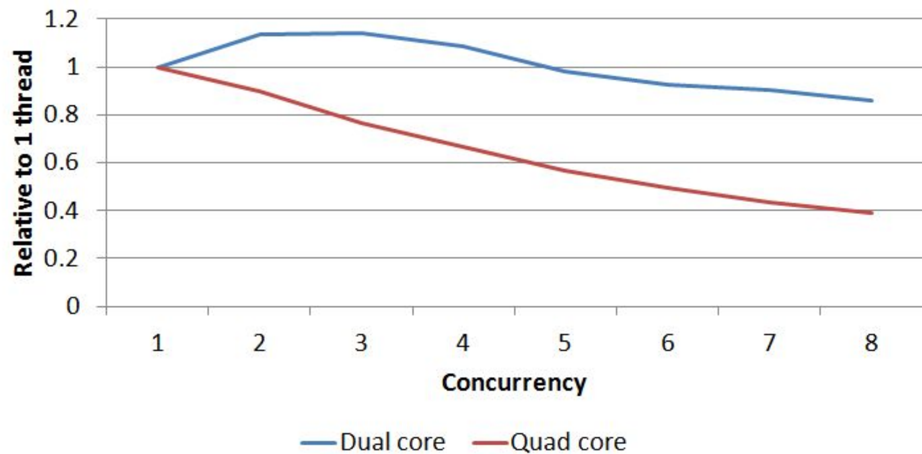
1. ~~Uses std::random_shuffle (deprecated)~~
2. Lock degrade currently racy under multiple thread users of same object instance
3. Networked user lock degrade blocks all lock user until all users realise degrade is happening
4. Multiple object instances racy on POSIX systems with insane byte locks

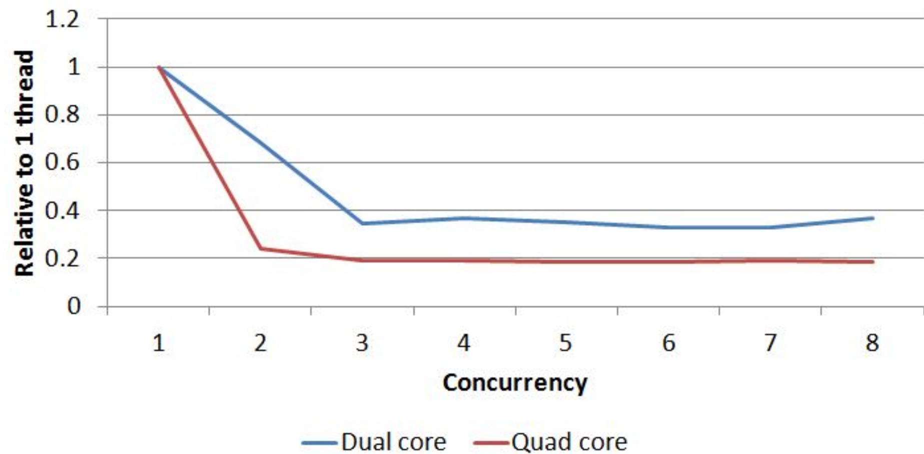

Scaling of two contended users of the lock algorithms to entities on NTFS
100000000
10000000
1000000
100000
10000
1000
100
Max ops/sec
1
2
3
4
5
6
7
8
Entities
lock files
byte ranges
atomic append
memory map

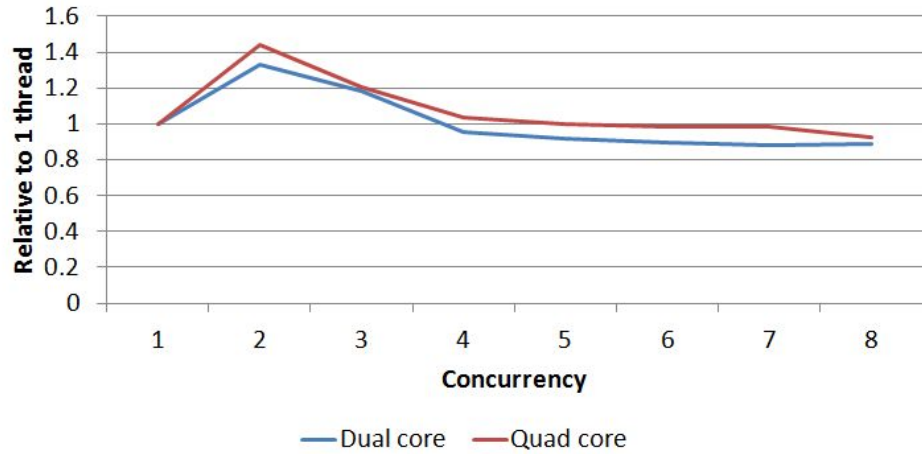

Scaling of four concurrent users of the lock algorithms to entities on NTFS
100000000
10000000
1000000
100000
10000
1000
100
Max ops/sec
1
2
3
4
5
6
7
8
Entities
lock files
byte ranges
atomic append
memory map

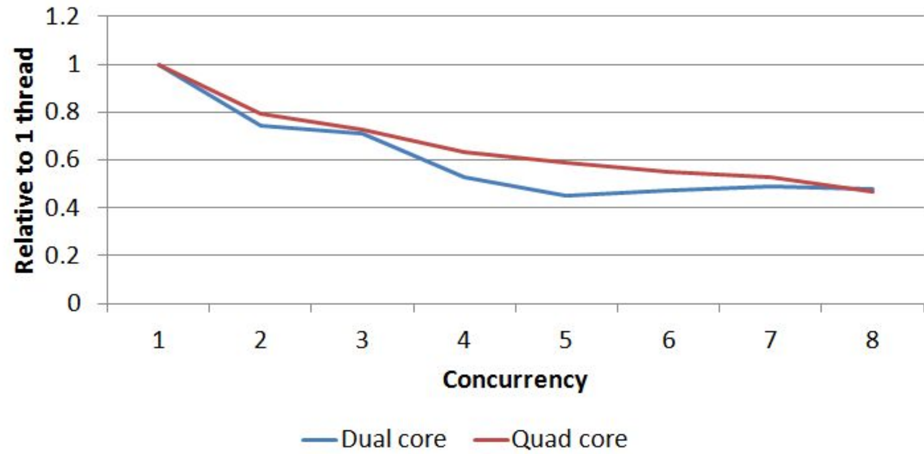
Concurrency scaling

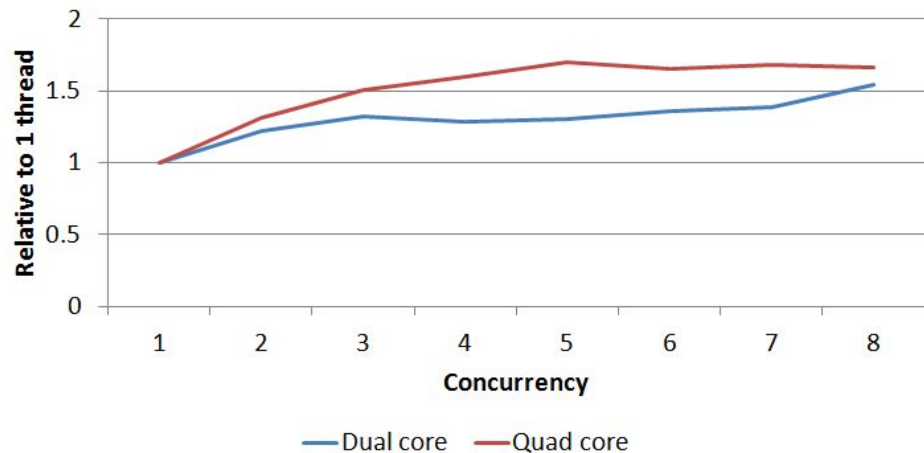
Questions before we begin this section?

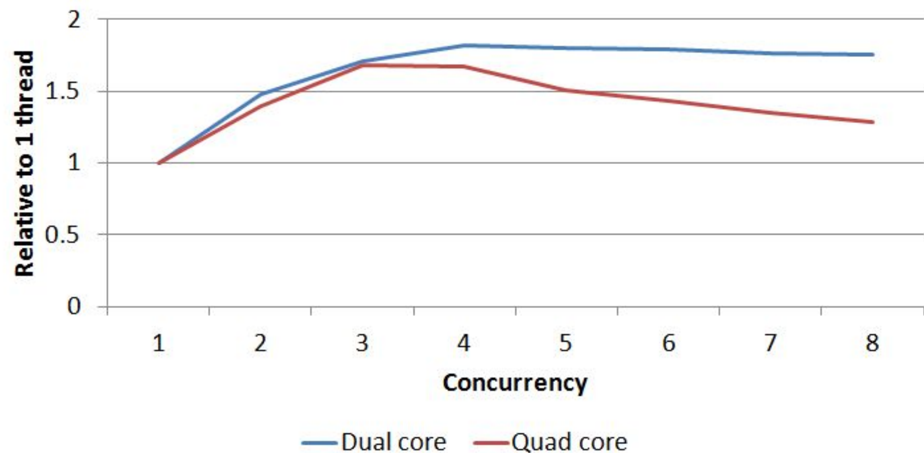

Contended lock files
Relative to 1 thread
1.2
1
0.8
0.6
0.4
0.2
0
1
2
3
4
5
6
7
8
Concurrency
Dual core
Quad core

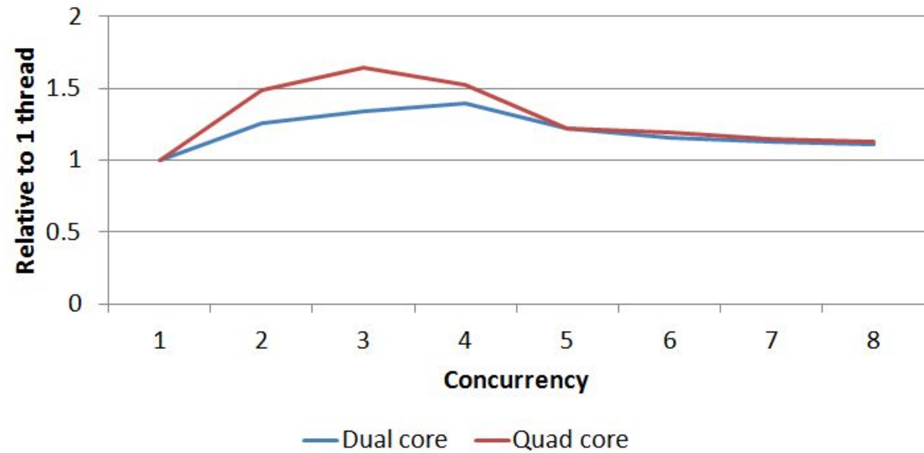

Contended byte ranges
Relative to 1 thread
1.2
1
0.8
0.6
0.4
0.2
0
1
2
3
4
5
6
7
8
Concurrency
Dual core
Quad core

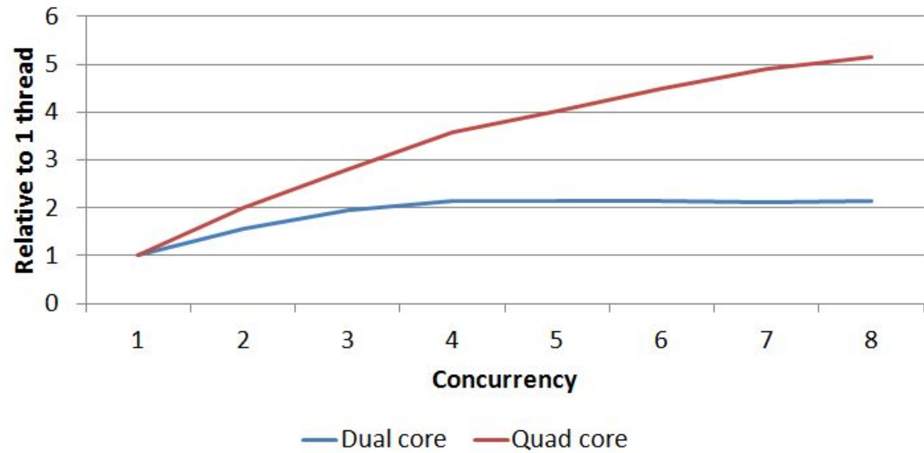

Contended atomic append
Relative to 1 thread
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
1
2
3
4
5
6
7
8
Concurrency
Dual core
Quad core


Contended memory maps
Relative to 1 thread
1.2
1
0.8
0.6
0.4
0.2
0
1
2
3
4
5
6
7
8
Concurrency
Dual core
Quad core


Uncontended lock files
Relative to 1 thread
2
1.5
1
0.5
0
1
2
3
4
5
6
7
8
Concurrency
Dual core
Quad core


Uncontended byte ranges
Relative to 1 thread
2
1.5
1
0.5
0
1
2
3
4
5
6
7
8
Concurrency
Dual core
Quad core


Uncontended atomic append
Relative to 1 thread
2
1.5
1
0.5
0
1
2
3
4
5
6
7
8
Concurrency
Dual core
Quad core


Uncontended memory maps
Relative to 1 thread
6
5
4
3
2
1
0
1
2
3
4
5
6
7
8
Concurrency
Dual core
Quad core

NTFS

lock_files

	1 entities	2 entities	3 entities	4 entities	5 entities	6 entities	7 entities	8 entities
1 waiters	3225	1548	979	773	622	522	448	404
2 waiters	3663	1717	1058	749	660	499	333	173
3 waiters	3676	1559	869	556	299	177	119	88
4 waiters	3507	1456	468	265	155	104	83	60
5 waiters	3170	1298	326	203	115	82	57	43
6 waiters	2993	958	282	161	89	62	45	31
7 waiters	2921	896	264	123	78	48	31	25
8 waiters	2771	524	221	123	62	44	25	19

byte_ranges

	1 entities	2 entities	3 entities	4 entities	5 entities	6 entities	7 entities	8 entities
1 waiters	577811	296029	197740	149256	107958	100535	85732	75138
2 waiters	395879	142159	120077	58698	87180	76342	45294	25083
3 waiters	201341	119729	120751	111594	95092	79542	54863	28078
4 waiters	212394	113706	83977	81919	29144	32086	10601	4480
5 waiters	202346	110786	108678	97624	71422	15104	3508	2844
6 waiters	188840	93870	84291	90080	24534	4261	2642	2195
7 waiters	191017	104915	100005	68568	12964	3057	2311	2136
8 waiters	212472	103827	58576	23138	4123	2670	2249	2143

atomic_append

	1 entities	2 entities	3 entities	4 entities	5 entities	6 entities	7 entities	8 entities
1 waiters	62585	62207	61542	62805	62366	61434	62603	62119
2 waiters	83489	85748	81203	79872	70425	76958	77422	81651
3 waiters	74157	74859	74343	74380	74604	75360	74574	75318
4 waiters	59858	59977	60647	59955	60022	58904	60003	59328
5 waiters	57662	57868	58221	58120	57901	57932	57594	57948
6 waiters	56053	55935	56111	55512	55806	55998	55914	56229
7 waiters	55299	55422	55609	55385	55472	55213	55104	55325
8 waiters	55456	55129	55425	55370	55438	55720	55477	55741

memory_map

	1 entities	2 entities	3 entities	4 entities	5 entities	6 entities	7 entities	8 entities
1 waiters	1E+07	8736749	7359580	6164645	5357319	4527427	3786885	2689792
2 waiters	7489908	6526982	5453116	4565350	3886815	3296295	2907728	2301558
3 waiters	7149246	5318787	5725922	3891240	3287970	2862335	2514230	2080585
4 waiters	5314168	4501485	3892188	3276682	2868544	2483889	2183352	1845890
5 waiters	4531635	3899585	3678885	3126871	2702327	2408430	2091723	1555216
6 waiters	4764991	4040055	3790918	3299163	2750590	2459851	2149692	1543167
7 waiters	4933737	4136594	3872380	3309020	2805250	2470896	2170246	1559174
8 waiters	4841813	4253667	3967508	3358823	2861877	2497987	2180971	1567126

!lock_files

	1 entities	2 entities	3 entities	4 entities	5 entities	6 entities	7 entities	8 entities
1 waiters	2909	1438	967	745	585	455	407	350
2 waiters	3550	1701	1202	872	705	588	475	430
3 waiters	3834	1804	1206	910	705	589	528	441
4 waiters	3742	1884	1204	906	746	608	510	440
5 waiters	3806	1901	1267	941	753	631	539	466
6 waiters	3951	1958	1373	972	777	631	545	495
7 waiters	4046	2029	1357	1017	813	640	549	470
8 waiters	4496	2014	1402	1020	777	677	543	517

!byte_ranges

	1 entities	2 entities	3 entities	4 entities	5 entities	6 entities	7 entities	8 entities
1 waiters	591620	297724	201182	151854	105881	101090	84476	76541
2 waiters	873678	443791	287517	213171	169548	138431	120790	105809
3 waiters	1013366	493416	316390	227890	183447	151368	129916	114330
4 waiters	1077461	535494	348619	245531	197498	155970	138553	118115
5 waiters	1066335	528127	341497	250788	192878	158116	137599	115318
6 waiters	1062478	504259	337299	250581	191613	160244	130926	118840
7 waiters	1044140	508833	336061	246945	186329	157049	136279	114955
8 waiters	1039428	511027	335178	247347	189047	153060	132999	118271

!atomic_append

	1 entities	2 entities	3 entities	4 entities	5 entities	6 entities	7 entities	8 entities
1 waiters	62262	63195	63276	62959	63238	62377	63595	63185
2 waiters	78468	80924	77032	82316	82866	82033	84138	75966
3 waiters	83437	83374	82644	81890	83382	82922	83060	83705
4 waiters	86871	87444	87234	89427	85681	88254	87496	88889
5 waiters	76252	77940	75874	76025	76158	77174	76622	77022
6 waiters	71977	70866	72501	71940	72910	71759	72147	72493
7 waiters	70149	69809	69293	70793	70349	70682	68998	69410
8 waiters	69008	68578	64779	66854	67189	67828	69632	68405

!memory_map

	1 entities	2 entities	3 entities	4 entities	5 entities	6 entities	7 entities	8 entities
1 waiters	1.1E+07	9369366	7489474	6377522	5225148	4557502	3900521	2710251
2 waiters	1.7E+07	1.4E+07	1.2E+07	1E+07	8532551	7267296	6188444	4545191
3 waiters	2.1E+07	1.8E+07	1.5E+07	1.2E+07	1E+07	8878937	7668981	5870409
4 waiters	2.3E+07	1.9E+07	1.6E+07	1.3E+07	1.1E+07	9553637	8334436	6749362
5 waiters	2.3E+07	1.9E+07	1.6E+07	1.3E+07	1.1E+07	9638724	8336666	6749031
6 waiters	2.3E+07	1.9E+07	1.6E+07	1.3E+07	1.1E+07	9602251	8301431	6757688
7 waiters	2.3E+07	1.9E+07	1.6E+07	1.3E+07	1.1E+07	9637157	8251208	6770029
8 waiters	2.3E+07	1.8E+07	1.6E+07	1.3E+07	1.1E+07	9623359	8276757	6717221

ReFS

lock_files

	1 entities	2 entities	3 entities	4 entities	5 entities	6 entities	7 entities	8 entities
1 waiters	1778	885	575	433	337	289	239	193
2 waiters	1721	856	274	172	137	96	82	75
3 waiters	1343	817	230	128	95	64	53	39
4 waiters	1350	746	185	108	60	53	32	29
5 waiters	1251	702	167	77	47	36	25	15
6 waiters	1170	396	135	66	38	22	15	14
7 waiters	1119	420	116	56	39	16	19	14
8 waiters	1077	263	108	54	26	15	15	10

byte_ranges

	1 entities	2 entities	3 entities	4 entities	5 entities	6 entities	7 entities	8 entities
1 waiters	1028550	520213	356183	266872	199491	174761	131250	132080
2 waiters	147105	95502	49844	98065	94209	25537	34731	22051
3 waiters	211325	48373	151711	111803	74199	78943	68602	72383
4 waiters	167487	59099	58536	36708	32224	22506	22495	60953
5 waiters	220012	170473	128873	76189	30585	67190	46694	46703
6 waiters	88740	52941	44554	43073	91770	47076	50262	19321
7 waiters	203785	161033	127959	119133	81924	61040	22758	4109
8 waiters	84749	57221	55076	39044	79235	40358	5568	1739

atomic_append

	1 entities	2 entities	3 entities	4 entities	5 entities	6 entities	7 entities	8 entities
1 waiters	73918	73721	73507	73331	73902	73950	74353	73480
2 waiters	69587	53963	53596	55371	57507	52513	41748	52999
3 waiters	55658	52130	55622	52074	54549	55827	52924	56233
4 waiters	39485	40164	40753	40994	40163	41035	40735	41052
5 waiters	37040	37418	37183	37406	37250	37177	37011	37435
6 waiters	36234	36668	36250	36379	36148	36657	36232	36628
7 waiters	36000	36259	36216	36182	36083	36361	35936	36351
8 waiters	35151	35947	35440	35723	35414	35839	35638	35755

!lock_files

	1 entities	2 entities	3 entities	4 entities	5 entities	6 entities	7 entities	8 entities
1 waiters	1761	899	592	418	349	282	247	217
2 waiters	2156	1003	707	514	419	336	297	258
3 waiters	2150	1103	692	534	424	338	293	265
4 waiters	2224	1112	698	523	430	357	290	266
5 waiters	2220	1119	735	523	435	341	305	267
6 waiters	2353	1078	736	531	439	360	302	271
7 waiters	2372	1068	748	541	444	358	310	275
8 waiters	2348	1128	754	560	421	370	314	259

!byte_ranges

	1 entities	2 entities	3 entities	4 entities	5 entities	6 entities	7 entities	8 entities
1 waiters	1045024	520411	355979	266540	211000	148776	151066	131804
2 waiters	1477402	528860	135038	145839	235155	154452	90226	35991
3 waiters	194610	77004	35954	38425	25111	50603	43209	15568
4 waiters	132911	61955	119189	64402	40900	47368	27307	16595
5 waiters	350358	197264	92625	95750	66667	38116	43134	36594
6 waiters	332401	195857	87038	64157	54242	58424	38474	37733
7 waiters	333198	200036	129953	95895	58470	48473	51701	39208
8 waiters	427728	193048	127801	94760	74075	60952	42841	36861

!atomic_append

	1 entities	2 entities	3 entities	4 entities	5 entities	6 entities	7 entities	8 entities
1 waiters	73684	73841	73987	73157	74479	74293	73735	74128
2 waiters	59973	73969	64406	62279	77340	61615	66668	51723
3 waiters	69173	68034	60841	68588	68679	66559	68932	68809
4 waiters	63048	63258	63254	63073	63123	63038	63094	62986
5 waiters	55039	55300	55353	55254	55513	55300	55268	54761
6 waiters	55307	55666	55759	54260	55286	55179	55352	55362
7 waiters	55676	56561	56260	56486	55704	56395	56323	56190
8 waiters	56381	56457	57552	57459	57183	57317	56779	56886

FAT32

lock_files

	1 entities	2 entities	3 entities	4 entities	5 entities	6 entities	7 entities	8 entities
1 waiters	4126	2050	1337	1007	797	663	566	495
2 waiters	3702	1972	1118	590	443	359	296	251
3 waiters	3258	1559	619	370	256	202	168	126
4 waiters	2967	991	390	235	153	108	86	68
5 waiters	2756	708	255	151	101	66	46	35
6 waiters	2613	479	208	111	63	41	35	23
7 waiters	2476	411	150	78	46	34	24	17
8 waiters	2330	364	124	60	34	25	16	12

byte_ranges

	1 entities	2 entities	3 entities	4 entities	5 entities	6 entities	7 entities	8 entities
1 waiters	1297963	649646	439432	333246	232784	221984	188825	165264
2 waiters	257238	67785	64820	111818	144412	115606	67078	23677
3 waiters	66439	53225	150851	154250	138002	119058	102922	109026
4 waiters	168795	65317	57760	49707	48906	39939	38234	31888
5 waiters	230486	184083	61486	145454	132486	114239	107321	78113
6 waiters	174589	103261	86641	46551	45428	43764	83868	13660
7 waiters	226353	173508	163148	147564	128971	91736	47306	6149
8 waiters	151525	58934	51814	52859	50787	61537	6977	3293

atomic_append

	1 entities	2 entities	3 entities	4 entities	5 entities	6 entities	7 entities	8 entities
1 waiters	87793	87324	86993	88028	87932	87852	87492	87693
2 waiters	113249	111361	105834	104316	114361	112666	80450	78941
3 waiters	79563	79156	79160	78737	79699	79638	79329	79760
4 waiters	64640	64628	65525	64893	65592	64684	65234	64766
5 waiters	63913	63808	63786	63654	63184	63520	63642	63646
6 waiters	61629	61672	61770	61809	61484	61505	61514	61493
7 waiters	60468	60683	60708	60742	60722	60646	60779	61090
8 waiters	60108	60196	59838	60380	60454	60507	60290	59380

!lock_files

	1 entities	2 entities	3 entities	4 entities	5 entities	6 entities	7 entities	8 entities
1 waiters	3845	1982	1294	972	774	641	550	473
2 waiters	5219	2563	1693	1262	998	834	713	621
3 waiters	5826	2688	1880	1393	1114	928	794	672
4 waiters	6044	2978	1963	1456	1166	962	815	713
5 waiters	6298	3075	2027	1509	1191	992	840	735
6 waiters	6373	3141	2066	1526	1213	999	861	740
7 waiters	6467	3151	2085	1553	1232	1025	864	747
8 waiters	6432	3187	2105	1561	1240	1018	862	746

!byte_ranges

	1 entities	2 entities	3 entities	4 entities	5 entities	6 entities	7 entities	8 entities
1 waiters	1325742	651134	445110	335650	235706	221729	190568	141174
2 waiters	1680911	792593	506289	370064	284546	225984	159069	153741
3 waiters	1766110	807595	485385	338797	276625	221715	182034	158496
4 waiters	1733640	748291	445838	306652	230967	189784	157505	136059
5 waiters	1672027	699749	426576	297692	228880	179782	153433	135206
6 waiters	1620083	690979	414241	291869	215476	178984	153737	131952
7 waiters	1597822	675096	414863	290919	223037	177217	151001	132833
8 waiters	1582453	661316	406077	285991	215080	177074	153112	130403

!atomic_append

	1 entities	2 entities	3 entities	4 entities	5 entities	6 entities	7 entities	8 entities
1 waiters	87128	87425	87594	87810	87317	87427	87141	86341
2 waiters	89044	112755	113221	114158	112896	114243	115829	112943
3 waiters	101447	102457	101626	102488	102042	102238	101394	101496
4 waiters	94863	95024	95791	96045	95350	95189	95761	95345
5 waiters	82169	81923	81788	81901	81746	80855	81572	81428
6 waiters	80914	76546	79837	81304	80305	80403	79953	80044
7 waiters	80532	81554	81143	80894	80464	75865	79965	79507
8 waiters	79782	81153	81920	81259	79825	79079	80158	79793

exFAT

lock_files

	1 entities	2 entities	3 entities	4 entities	5 entities	6 entities	7 entities	8 entities
1 waiters	3726	1690	1098	819	651	543	469	410
2 waiters	2912	1625	810	397	315	252	213	185
3 waiters	2558	1207	367	231	178	136	113	98
4 waiters	2190	737	194	142	105	78	60	67
5 waiters	1996	551	141	93	59	46	35	32
6 waiters	1851	452	103	65	39	33	20	16
7 waiters	1695	215	81	45	27	19	12	12
8 waiters	1575	327	62	34	17	14	11	5

byte_ranges

	1 entities	2 entities	3 entities	4 entities	5 entities	6 entities	7 entities	8 entities
1 waiters	1276972	638981	437970	320686	255046	180394	187836	165060
2 waiters	303808	144943	83696	111243	136074	66560	33090	25279
3 waiters	235183	181299	157596	145089	107825	108396	31487	104735
4 waiters	87896	62657	67041	48958	50980	37125	37315	29180
5 waiters	219985	188574	127549	44749	134631	116242	91379	83509
6 waiters	86217	89597	108789	81226	52096	32584	34573	7475
7 waiters	227004	176342	159881	143632	126400	91463	38449	4589
8 waiters	81512	54944	50445	48631	51229	31016	4036	2663

atomic_append

	1 entities	2 entities	3 entities	4 entities	5 entities	6 entities	7 entities	8 entities
1 waiters	85689	86584	86522	86634	86151	87054	86466	87020
2 waiters	89396	114035	112316	89724	61888	89214	83526	116452
3 waiters	79858	80587	80396	80665	80088	80326	79865	79538
4 waiters	65129	64532	60795	63315	64987	64594	64710	65255
5 waiters	63975	61311	63435	63464	63318	63731	62666	63681
6 waiters	61740	61406	61574	61226	61363	61838	61466	61717
7 waiters	60873	60813	60391	60413	60442	60854	60754	60112
8 waiters	59610	59996	60264	59609	60102	59943	60156	59861

!lock_files

	1 entities	2 entities	3 entities	4 entities	5 entities	6 entities	7 entities	8 entities
1 waiters	3254	1624	1065	802	643	534	458	399
2 waiters	4510	2152	1446	1073	855	710	606	528
3 waiters	5007	2404	1592	1178	929	788	655	569
4 waiters	5299	2595	1688	1258	988	817	696	602
5 waiters	5536	2678	1773	1314	1037	847	716	623
6 waiters	5741	2749	1816	1342	1056	870	731	632
7 waiters	5832	2786	1792	1358	1076	885	744	641
8 waiters	5869	2849	1872	1385	1085	882	741	643

!byte_ranges

	1 entities	2 entities	3 entities	4 entities	5 entities	6 entities	7 entities	8 entities
1 waiters	1280670	656689	437435	323985	251656	199019	167644	163807
2 waiters	1658985	780909	486963	362693	287461	214017	198650	155709
3 waiters	1733442	795368	495761	356799	256000	216859	165515	160322
4 waiters	1701304	740913	441650	299441	229149	182145	153369	134874
5 waiters	1638192	701754	425566	288508	213805	179101	154379	131910
6 waiters	1616108	691817	417782	288313	222022	176361	150580	132539
7 waiters	1551642	665967	407890	287003	212915	175691	151625	129728
8 waiters	1541039	647714	401734	279177	215794	172387	146291	129103

!atomic_append

	1 entities	2 entities	3 entities	4 entities	5 entities	6 entities	7 entities	8 entities
1 waiters	86370	87509	87204	86897	87233	86852	87499	87161
2 waiters	109122	100898	112204	94047	92625	94208	103463	113740
3 waiters	100335	102446	101729	101960	101811	101260	102097	101015
4 waiters	96164	96415	96510	96234	95838	96418	96236	95848
5 waiters	81547	82173	81938	81945	81967	81865	81579	81407
6 waiters	81341	81183	81335	81370	79920	80917	80867	80792
7 waiters	80628	81399	80443	80298	80788	81137	80965	79777
8 waiters	80423	81375	81602	80768	81155	79876	80585	79766

Excel spreadsheet is in /graphs in AFIO git repo

Thank you

And let the questions begin!

Github: https://github.com/ned14/boost.afio

Ref docs: https://ned14.github.io/boost.afio/