POLARDB 数据库性能大赛总决赛

“至尊宝”战队答辩

Outline

- Problem Anatomy
- Measurements
- Designs an Implementations
- Iterations and Results
- Beyond the Results
- Innovations
- Appendix

- Basic Constraints
  - Memory limited
    - State limited (2GB for C++)
    - Input must be stored into disk
      - EXT4 needed(Raw disk is not allowed)
      - DIO wanted(Buffered IO is globally blocking and not controllable)
  - No compression
    - Good randoms hard to be compressed
    - Computing power overflow

Problem Anatomy

- Basic Constraints
  - Arbitrary Crash resisted
    - WAL somewhere?
      - Sync
      - Kernel managed buffer (file mmaped memeory)

- Basic Relaxations
  - Fixed length keys/values
  - Write and Read separated

Problem Anatomy

- *Goal: Striving to run at full IO bandwidth*
    - Required to implement: *Write/Read/Range*
    - (point) *Read*: special case of *Range*
- *Idea*
    - Design to make *Range* fastest
    - And then that design can be fastest in *Write/Read* as well

- *Idea(cont.)*
  - for *Range*
    - full range iteration - global order
    - How order?
    - Order of what?
    - When to order?

- *Idea(cont.)*
  - How order?
    - Global in one store(file)
      - expensive in (ext4) file system
    - inter-store order - range partition
      - + intra-store order -> global order

Problem Anatomy

- *Idea(cont.)*
  - Order of what?
    - Key (as least)
      - value is seq appended
    - Key + value
      - sub-optimal for modern (nvme) storage

- *Idea(cont.)*
  - When to order?
    - at building(Write) time
      - Take use of the redundant computing power @ write
    - at querying(Read/Range) time

Measurements

- *Start to code?*
- Measure, Measure, Measure
  - Not just blindly benchmark
  - Measurements make optimizations scientific (and help you further)

Basic Spec

Item	official spec[a]	public media
4k random write	500k	562,619[b-d]
4k random read	550k	580K[b-d]
sequential write	2000 MiB/s	2.17 GB/s[b-d]
sequential read	2400 MiB/s	2.53 GB/s(64K)[b-d]

a. https://ark.intel.com/products/97162/Intel-Optane-SSD-DC-P4800X-Series-375GB-1-2-Height-PCIe-x4-3D-XPoint-
b. https://www.tomshardware.com/reviews/intel-optane-3d-xpoint-p4800x,5030-5.html
c. https://www.storagereview.com/intel_optane_ssd_dc_p4800x_review
d. https://www.anandtech.com/show/11930/intel-optane-ssd-dc-p4800x-750gb-handson-review/3

Measurements

- *Basic Infos*
  - 4k random write/read throughput seem awesome if enough parallels
  - 4k random write on par with 4k sequential write
  - Sequential read faster than random read
  - Intel Device 2701(P4800x) controller is poor
    - 31 hardware IO queues
    - max 128k message
  - Intel Device 2700(900P) is cheaper version of 2701(P4800x)
    - Measurements from 900P will be shown later
  - Several configs interesting but can not be tweaked
    - IRQ affinity/polling/IO scheduler...

- *Hints*
  - *Write*
    - 4k-128k
    - 4k is unlikely(in that the contest)
  - *Read*
    - 64threads 4k random read may be enough
  - *Range*
    - Sequential read favored

Measurements

Is 4k enough if enough parallels(queue depth) (for Optane) in my work station with centos 7.2?

Item	official	public	Arch/4.19.x/EXT4	Cent OS 7.2/3.10.0-327/EXT4
4k random write	500k	562,619	552k	553k
4k random read	550k	580k	587k	587k
sequential write	2000 MiB/s	2.17 GB/s	2204 MiB/s	2207 MiB/s(16k bs, 564,992 4k eq)
sequential read	2400 MiB/s	2.53 GB/s	2607 MiB/s	2613 MiB/s(1M bs)

1. Hardware: 2*Xeon Gold 5120, Intel Device 2700(900P)/280G
2. peak records of fio 3.1x runs under all conditions, detailed conditions seen later and bench working flow seen in the crack guide

Measurements

```
[jin@bogon wen]$ fio --name=/wen/test_directory/fio_test_w -ioengine=psync -iode
pth=1 -rw=write -bs=16k -size=3G -direct=1 -numjobs=31 -runtime=100 -group_repor
ting -thread
/wen/test_directory/fio_test_w: (g=0): rw=write, bs=(R) 16.0KiB-16.0KiB, (W) 16.
0KiB-16.0KiB, (T) 16.0KiB-16.0KiB, ioengine=psync, iodepth=1
...
fio-3.11
Startin
/wen/tes
```

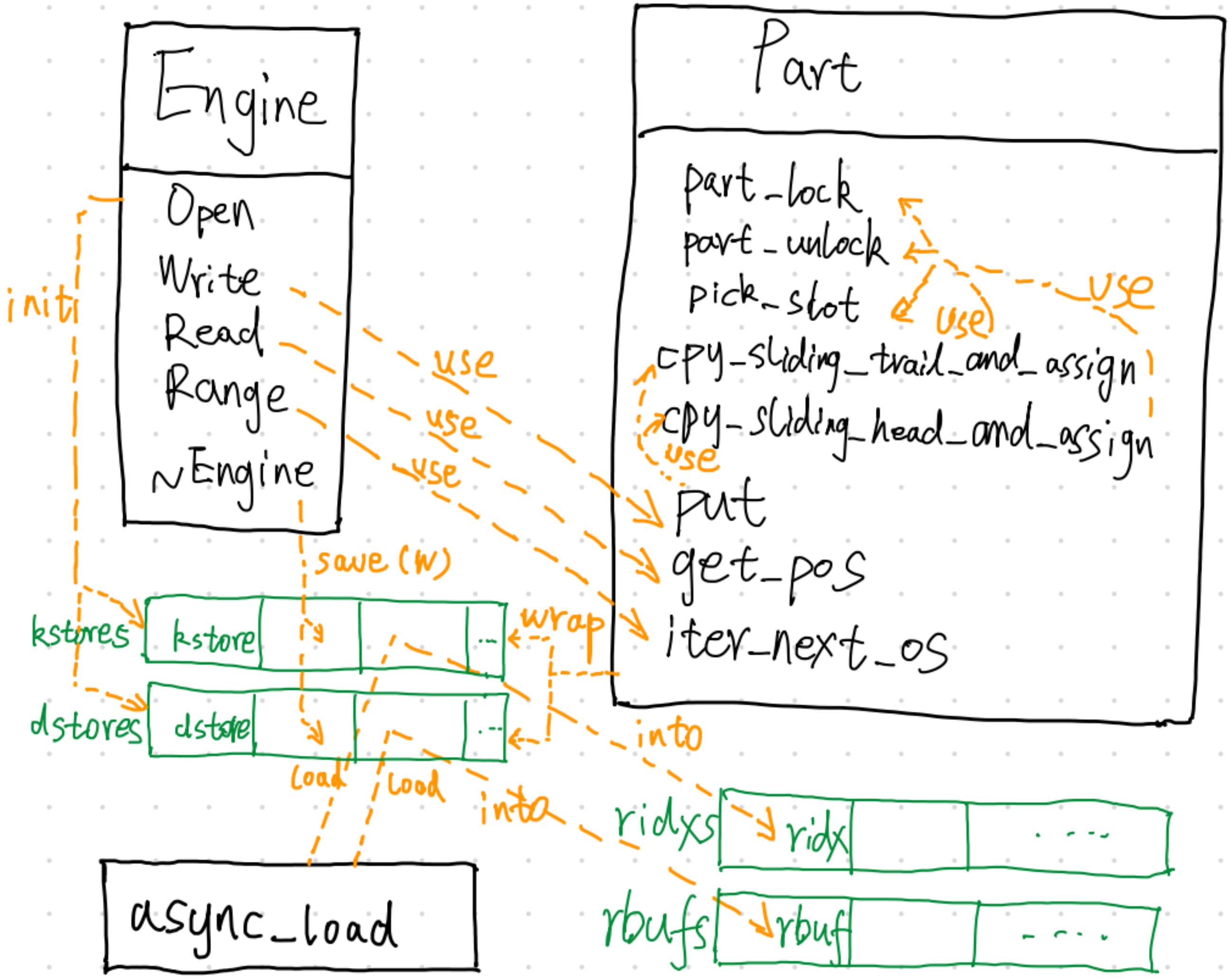
```
   bw (  KiB/s): min=67840, max=84160, per=3.23%, avg=72973.48, stdev=876.11, sa
mples=2652
   iops        : min= 4240, max= 5260, avg=4560.79, stdev=54.76, samples=2652
  lat (usec)   : 50=0.02%, 100=0.18%, 250=82.69%, 500=17.11%, 750=0.01%
  lat (usec)   : 1000=0.01%
  lat (msec)   : 2=0.01%, 10=0.01%, 20=0.01%, 50=0.01%
  cpu          : usr=1.81%, sys=17.43%, ctx=6167597, majf=0, minf=27
  IO depths    : 1=100.0%, 2=0.0%, 4=0.0%, 8=0.0%, 16=0.0%, 32=0.0%, >=64=0.0%
     submit    : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
     complete  : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
     issued rwts: total=0,6094848,0,0 short=0,0,0,0 dropped=0,0,0,0
     latency   : target=0, window=0, percentile=100.00%, depth=1

Run status group 0 (all jobs):
  WRITE: bw=2207MiB/s (2314MB/s), 2207MiB/s-2207MiB/s (2314MB/s-2314MB/s), io=93
.0GiB (99.9GB), run=43157-43157msec
```

- *Sum up*
  - Online benchmark environment(Cent OS 7.2) may have slightly better performance than my workstation.
    - Modern hardware+kernel greatly improve the perf of MMU activity (and 4k random write with libaio)
  - File system(EXT4) imposes more or less performance tax on disk device.
  - IO side decision:
    - *Write:* 4*4k bs, seq
    - *Read:* 4k bs, rand
    - *Range:* 1M bs, seq, 4Threads

- *Sum-up*
  - Lower Bound of Rank Scores from IO view
    - All peaks counts from measurements:

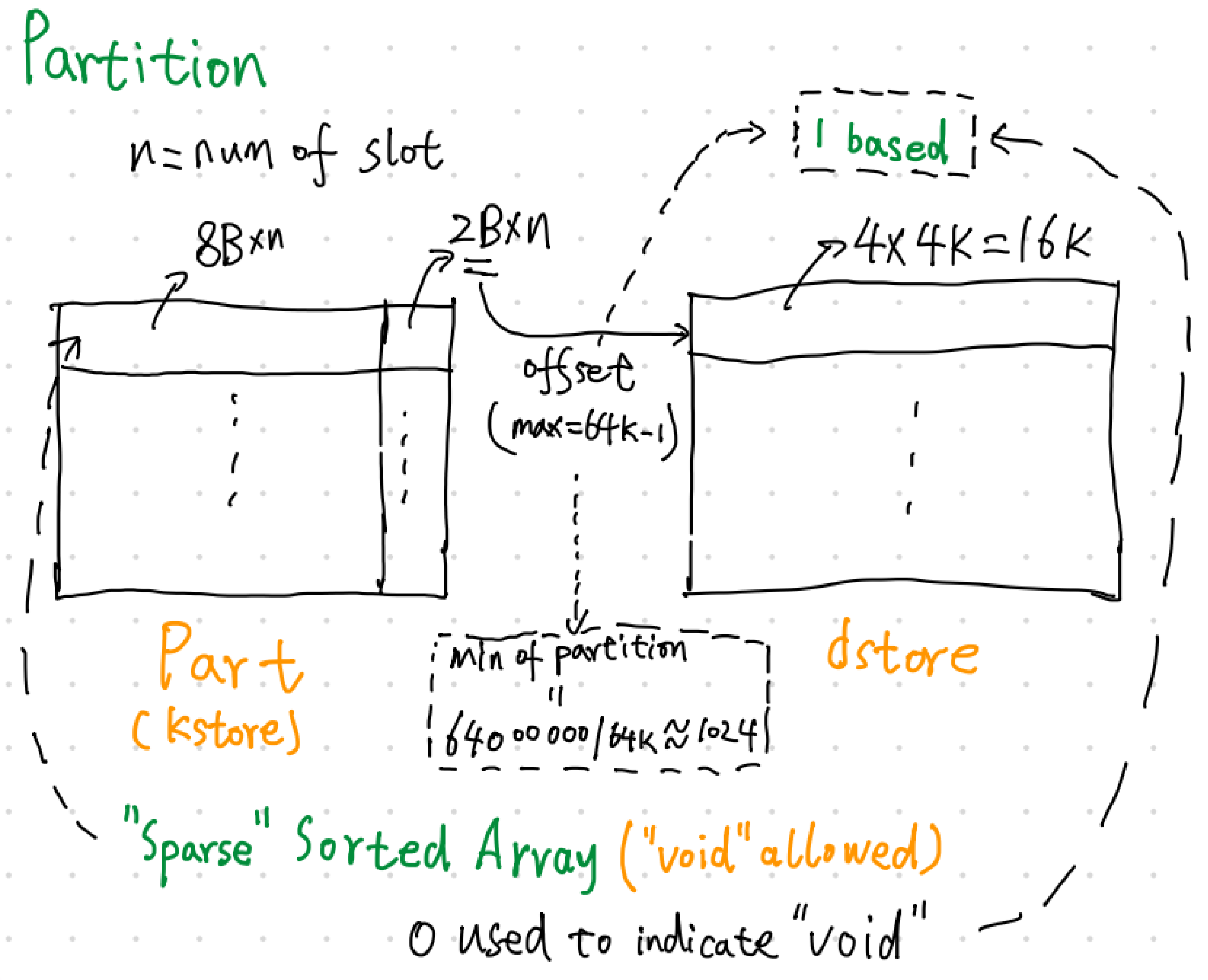
      64000000/564992+62000000/587000+2*64000000*4/(2613*1024)+0.7+0.9=411.85s
    - Relaxed to common top observations in online runs:

      64000000/562000+62000000/587000+2*64000000*4/(2601*1024)+0.7+0.9=413.41s
      - Write phase: only get one 562k(equ, online), common iops value is 558k
      - 64000000/558000-64000000/562000=0.816s
      - Range, in fact, has other tricks for improvement but which is considered anti-rule and meanless in engineering.

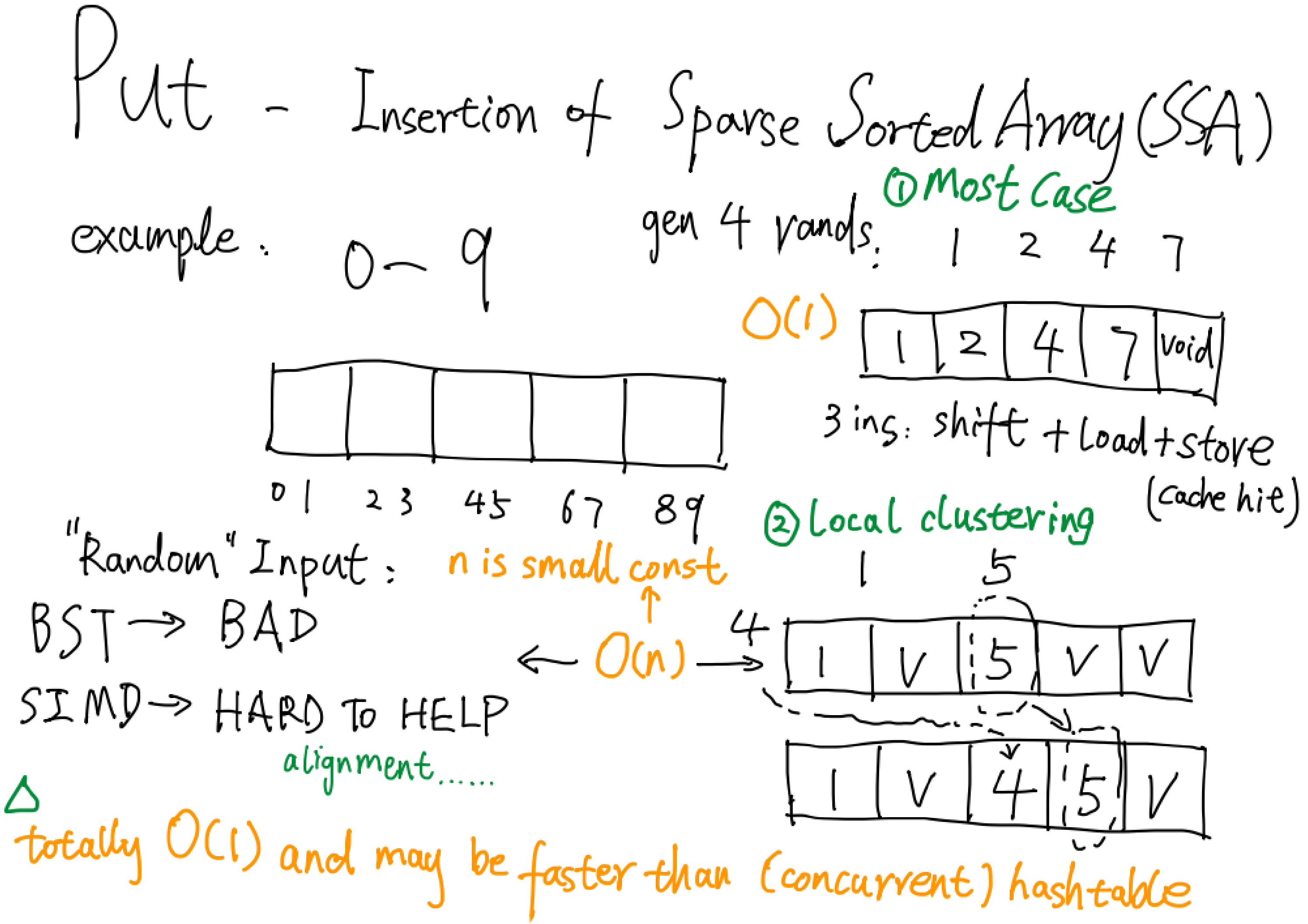


System design sketch

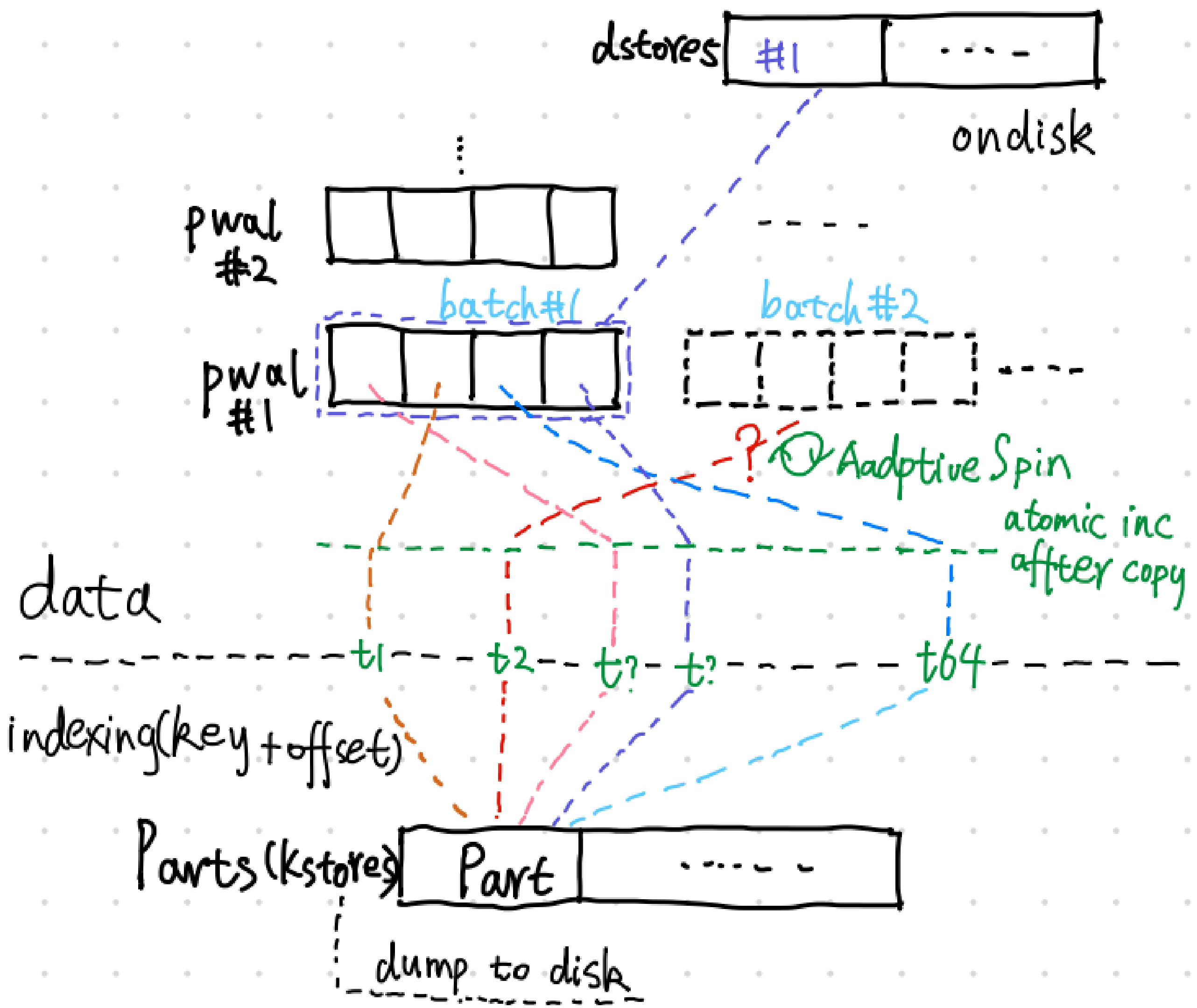
Keep It Simple Stupid



Part structure sketch



SSA insertion sketch

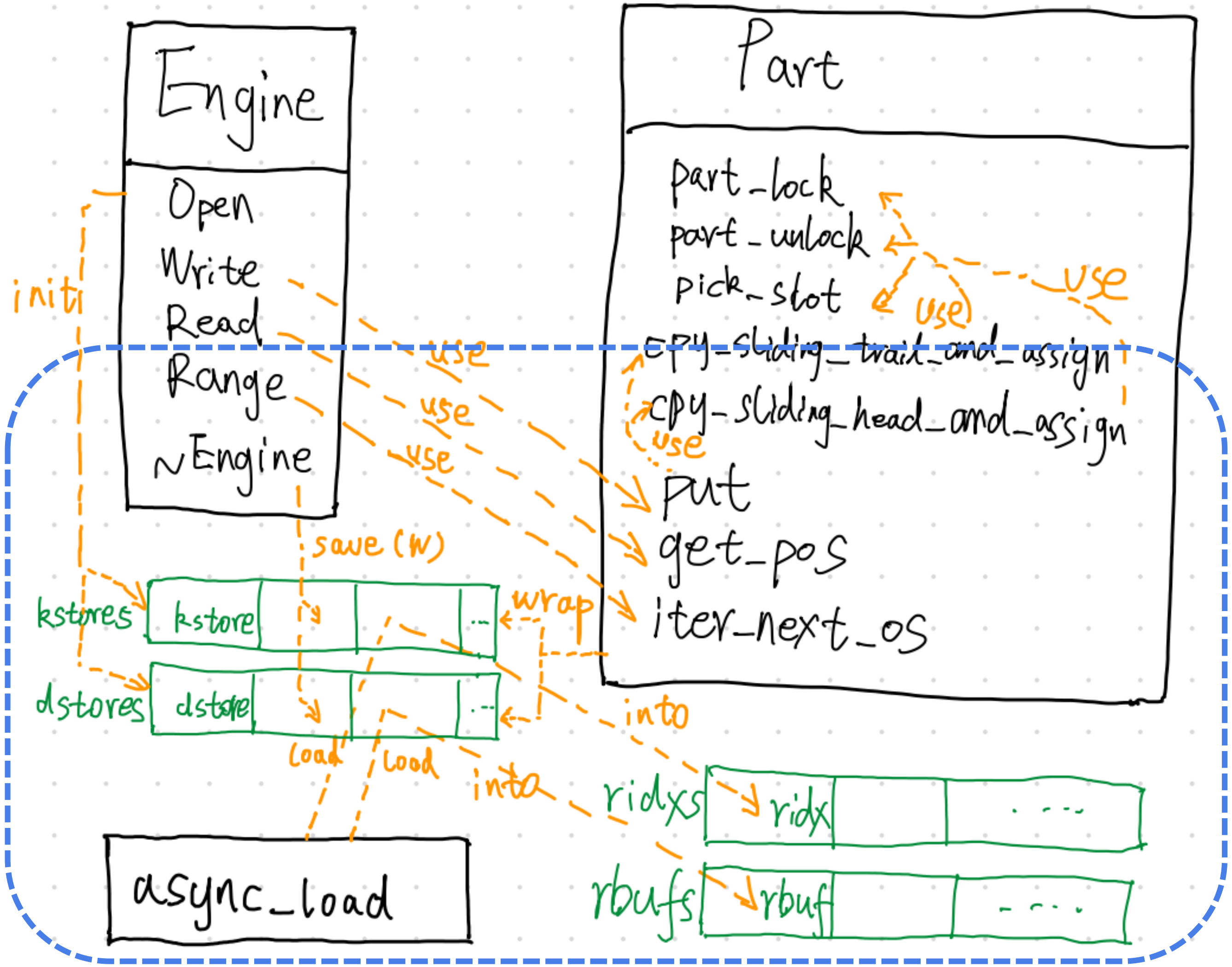


Logic of Write

Take use of pre-taken offset build a dump logic which is better than that of general lock-free queue

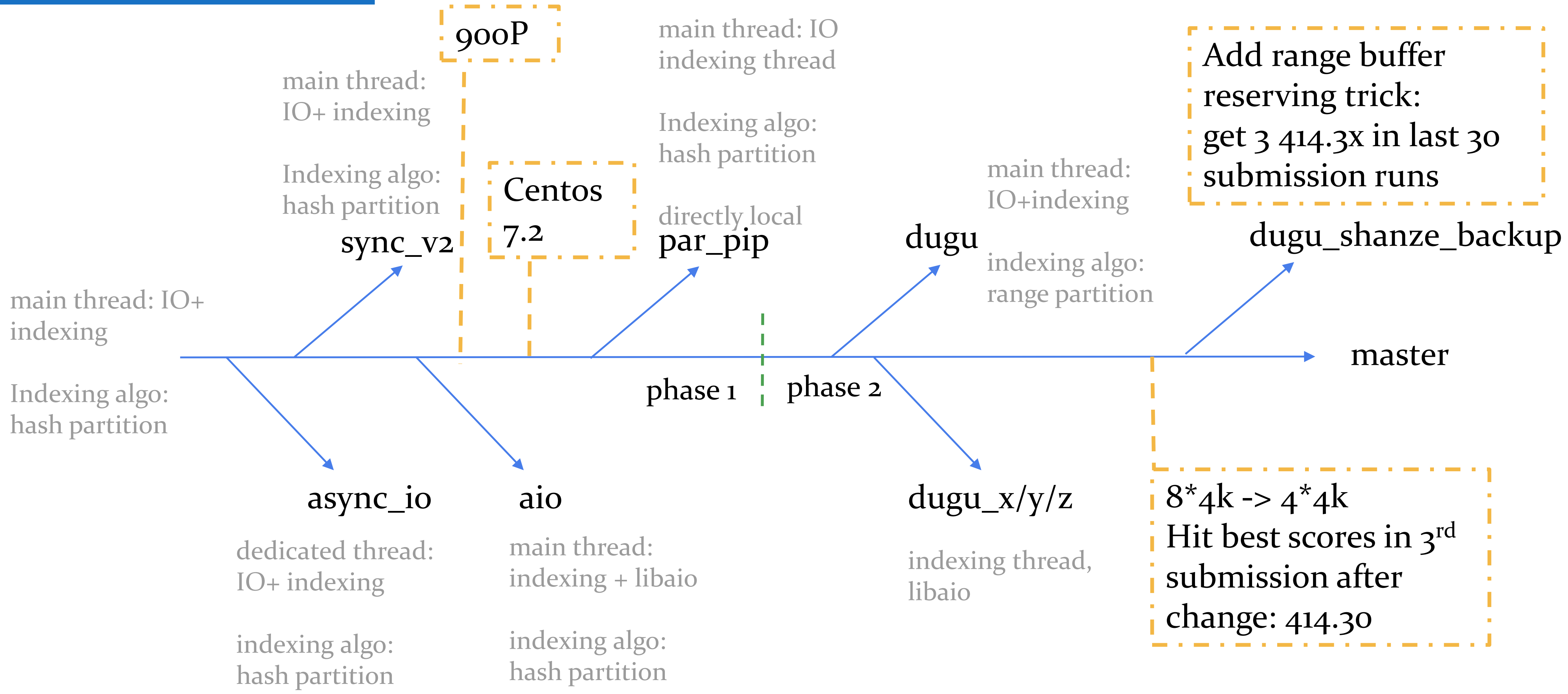
Logic of Read

Designs and Implementations



Logic of Range

Iterations and Results



- *Trick#1* range buffer reserving
  - 2 pass iterations
  - only use 1st part of range buffer slots in 1st pass to iterate in 2nd pass
  - only 2-4 minimum Part slots needed(4 slots locally tested, NOTE: Part obj size ~120MB)(if trick#2 used, then we only need 1 Part-sized slot)
  - definitely improve 0.7-0.9 seconds upon above general design(NOTE: mem limit of Java is 3G) if IO is assumed to be constant
  - done in dugu_shanze_backup branch
  - Outcome: 6/12 or 7/15, get 3 414.3x in last 30 submission runs
  - Ugly for engineering

- *Trick#2* single iterator
- iterates the values by single thread, then dispatches the values into a ring buffer, all clients (second-hand) consume extracted values
- crucial difference: the real time point to produce (and consume)
  - Idea of previous design works for arbitrary concurrent range iteration
  - But this trick can not in that it is just single iterator
    - advantage: no cache trashing
- owned online peak 42521287, others' 426xxxxx (1/425*200 ~= 0.47s)
  - this tiny diff also observed locally(by comparing fio and program runs)
- Not implemented (suspected violation to rule, nonsense for engineering)

Beyond the Results

- *Too many things beyond the results*
  - What's the effect of file system to fast storage?
  - How many modern features have we embraced?
  - What's better designs if more complex loads met?
  - How does the fast data(base) system evolve to its destination?

Beyond the Results

- File system
  - *"I can not reproduce the 4k random IOPS from public media."*
  - Investigation#1: Random IO IOPS is very poor for my Optane SSD when I mock the proposed benchmark flow via fio run.
    - Clean the test directory (and drop the cache) -> run -> ...

Beyond the Results

random write
slow down

-dsk/total-		--total-cpu-usage--					------memory-usage-----				--io/total-	
read	writ	usr	sys	idl	wai	stl	used	free	buff	cach	read	writ
1036k	37M	1	1	98	0	0	4184M	57.2G	456M	828M	14.7	7977
16k	27k	0	0	100	0	0	4184M	57.2G	456M	828M	0	4.00
0	81k	0	0	100	0	0	4185M	57.2G	456M	828M	0	8.33
1991k	0	0	1	99	0	0	4183M	57.2G	462M	839M	483	0
0	12k	0	0	100	0	0	4184M	57.2G	462M	839M	0	0.67
0	2252k	0	0	100	0	0	4185M	57.2G	463M	839M	0	18.0
853k	1123M	4	16	63	18	0	4586M	56.7G	482M	1245M	213	288k
0	2155M	4	30	30	35	0	4579M	56.6G	513M	1246M	0	552k
0	2149M	4	33	29	34	0	4574M	56.5G	539M	1246M	0	540k
0	2147M	4	33	29	34	0	4574M	56.4G	563M	1246M	0	533k
0	2146M	4	32	29	34	0	4750M	56.1G	584M	1246M	0	534k
0	1900M	3	31	43	23	0	4575M	56.2G	601M	1246M	0	417k
0	1716M	3	29	52	16	0	4580M	56.1G	615M	1246M	0	334k
0	1718M	3	29	52	16	0	4579M	56.1G	629M	1246M	0	335k
0	1672M	2	28	56	14	0	4576M	56.1G	641M	1246M	0	304k
0	1646M	3	30	54	13	0	4576M	56.0G	653M	1246M	0	314k
0	1619M	2	29	57	12	0	4575M	56.0G	664M	1246M	0	288k
0	1602M	3	28	59	11	0	4579M	56.0G	674M	1246M	0	284k
0	1634M	3	28	57	12	0	4585M	55.9G	683M	1246M	0	290k
0	1609M	2	28	58	11	0	4583M	55.9G	692M	1246M	0	286k
0	1602M	3	29	58	11	0	4585M	55.9G	700M	1246M	0	283k
0	1582M	2	28	60	10	0	4584M	55.9G	708M	1247M	0	271k

Beyond the Results

normal
seq write

dsk/total read	dsk/total writ	total-cpu-usage usr	sys	idl	wai	stl	memory-usage used	free	buff	cach	io/total read	io/total writ
1032k	41M	1	1	98	1	0	4304M	56.7G	814M	856M	14.6	8664
0	0	0	0	100	0	0	4303M	56.7G	814M	856M	0	0
0	1029M	3	8	69	20	0	4706M	56.3G	814M	1257M	0	264k
0	2149M	4	16	37	43	0	4702M	56.3G	814M	1258M	0	550k
0	2151M	4	17	36	43	0	4700M	56.3G	814M	1258M	0	550k
0	2152M	4	17	37	42	0	4696M	56.3G	815M	1258M	0	551k
0	2153M	4	16	37	43	0	4690M	56.3G	815M	1258M	0	551k
0	2158M	4	16	37	43	0	4688M	56.3G	815M	1258M	0	552k
0	2154M	4	16	37	43	0	4685M	56.3G	815M	1258M	0	551k
0	2157M	4	16	37	42	0	4682M	56.3G	815M	1258M	0	552k
0	2158M	4	16	37	43	0	4681M	56.3G	815M	1258M	0	552k
0	2153M	4	16	37	43	0	4679M	56.3G	815M	1258M	0	551k
0	2156M	4	16	37	43	0	4679M	56.3G	815M	1258M	0	552k
0	2155M	4	16	37	43	0	4678M	56.3G	815M	1258M	0	552k
0	2157M	4	16	37	43	0	4677M	56.3G	815M	1258M	0	552k
0	2152M	4	16	37	43	0	4677M	56.3G	815M	1258M	0	551k
0	2148M	4	16	37	43	0	4676M	56.3G	815M	1258M	0	550k
0	2147M	4	16	37	43	0	4677M	56.3G	815M	1258M	0	550k
0	2144M	4	16	37	43	0	4677M	56.3G	815M	1258M	0	549k
0	2144M	4	16	37	43	0	4677M	56.3G	815M	1258M	0	549k
0	2139M	4	16	37	43	0	4677M	56.3G	815M	1258M	0	548k
0	2139M	4	16	37	43	0	4676M	56.3G	815M	1258M	0	548k

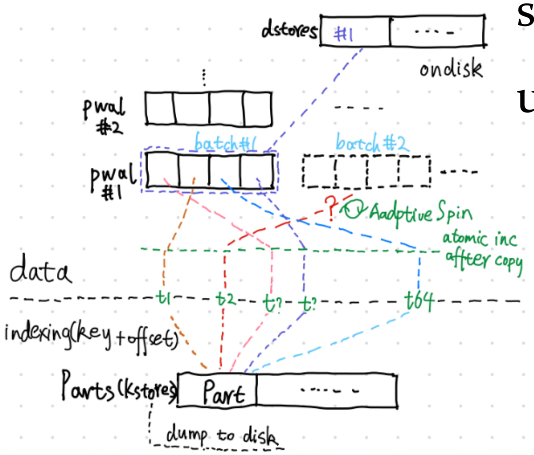
Beyond the Results

- *Why my online benchmark immune from this?*
  - *Random writes show huge overhead for un-pre-filled files writes. Forget it in the contest*
  - *fallocate cannot help for this*
  - *Just the sequential write*

Beyond the Results

- *Investigation#2:*
  - Program has a high probability (~2/3) to show an small unstable run

dsk/total read	dsk/total writ	cpu usr	cpu sys	cpu idl	cpu wai	cpu stl	mem used	mem free	mem buff	mem cach	io/total read	io/total writ
0	383M	3	7	84	5	0	5145M	51.7G	3783M	1967M	0	24.5k
0	2165M	4	8	37	51	0	5192M	51.6G	3783M	1967M	0	139k
0	1966M	6	18	31	46	0	5210M	51.6G	3785M	1973M	0	126k
0	2166M	4	8	35	53	0	5210M	51.6G	3785M	1967M	0	139k
0	1953M	5	18	31	47	0	5204M	51.6G	3788M	1968M	0	125k
0	2125M	4	14	33	49	0	5209M	51.6G	3788M	1967M	0	136k
0	1992M	4	9	35	52	0	5204M	51.6G	3790M	1967M	0	128k
0	1955M	4	17	32	48	0	5203M	51.6G	3793M	1967M	0	125k
0	2172M	4	8	35	53	0	5205M	51.6G	3793M	1967M	0	139k
0	1949M	4	17	32	47	0	5201M	51.6G	3796M	1967M	0	125k
0	2164M	4	7	36	53	0	5205M	51.6G	3796M	1967M	0	139k
0	1953M	4	17	32	47	0	5202M	51.6G	3798M	1967M	0	125k
0	2165M	4	8	35	53	0	5199M	51.6G	3798M	1967M	0	139k
0	1950M	4	17	32	48	0	5200M	51.6G	3801M	1967M	0	125k
0	1948M	4	16	33	48	0	5198M	51.6G	3803M	1967M	0	125k
0	2170M	4	8	36	52	0	5201M	51.6G	3803M	1967M	0	139k
0	1945M	4	17	32	47	0	5200M	51.6G	3806M	1967M	0	124k
0	2159M	4	8	36	52	0	5200M	51.6G	3806M	1967M	0	138k
0	1944M	4	17	32	48	0	5200M	51.6G	3808M	1967M	0	125k
0	2163M	4	8	35	53	0	5200M	51.6G	3808M	1967M	0	138k
0	1942M	4	17	35	44	0	5198M	51.6G	3811M	1967M	0	124k



Beyond the Results

0	905M	4	6	72	18	0	5160M	51.7G	3781M	1967M	0	58.0k
0	2188M	5	6	40	49	0	5176M	51.7G	3781M	1968M	0	140k
0	2185M	4	5	37	55	0	5175M	51.7G	3781M	1966M	0	140k
0	2187M	4	5	36	55	0	5172M	51.7G	3781M	1966M	0	140k
0	2191M	4	5	37	55	0	5173M	51.7G	3781M	1966M	0	140k
0	2183M	4	5	38	54	0	5173M	51.7G	3781M	1966M	0	140k
0	2186M	4	5	37	54	0	5173M	51.7G	3781M	1966M	0	140k
0	2185M	4	5	36	55	0	5173M	51.7G	3781M	1966M	0	140k
0	2186M	4	5	36	55	0	5173M	51.7G	3781M	1966M	0	140k
0	2183M	4	5	37	55	0	5174M	51.7G	3781M	1966M	0	140k
0	2186M	4	5	36	55	0	5174M	51.7G	3781M	1966M	0	140k
0	2187M	4	5	36	55	0	5174M	51.7G	3781M	1966M	0	140k
0	2189M	4	5	37	55	0	5173M	51.7G	3781M	1966M	0	140k
0	2188M	4	5	36	55	0	5173M	51.7G	3781M	1966M	0	140k
0	2190M	4	5	37	54	0	5173M	51.7G	3781M	1966M	0	140k
0	2194M	4	5	36	55	0	5173M	51.7G	3781M	1966M	0	140k
-dsk/total-		--total-cpu-usage--					------memory-usage-----				--io/total-	
read	writ	usr	sys	idl	wai	stl	used	free	buff	cach	read	writ
0	2204M	4	5	38	53	0	5173M	51.7G	3781M	1966M	0	141k
0	2200M	4	5	37	54	0	5173M	51.7G	3781M	1966M	0	141k
0	2202M	4	5	38	54	0	5173M	51.7G	3781M	1966M	0	141k
447M	1576M	3	4	69	24	0	5817M	51.0G	3781M	1966M	61.9k	88.9k
2293M	196k	5	6	41	48	0	5816M	51.0G	3781M	1966M	587k	2.67

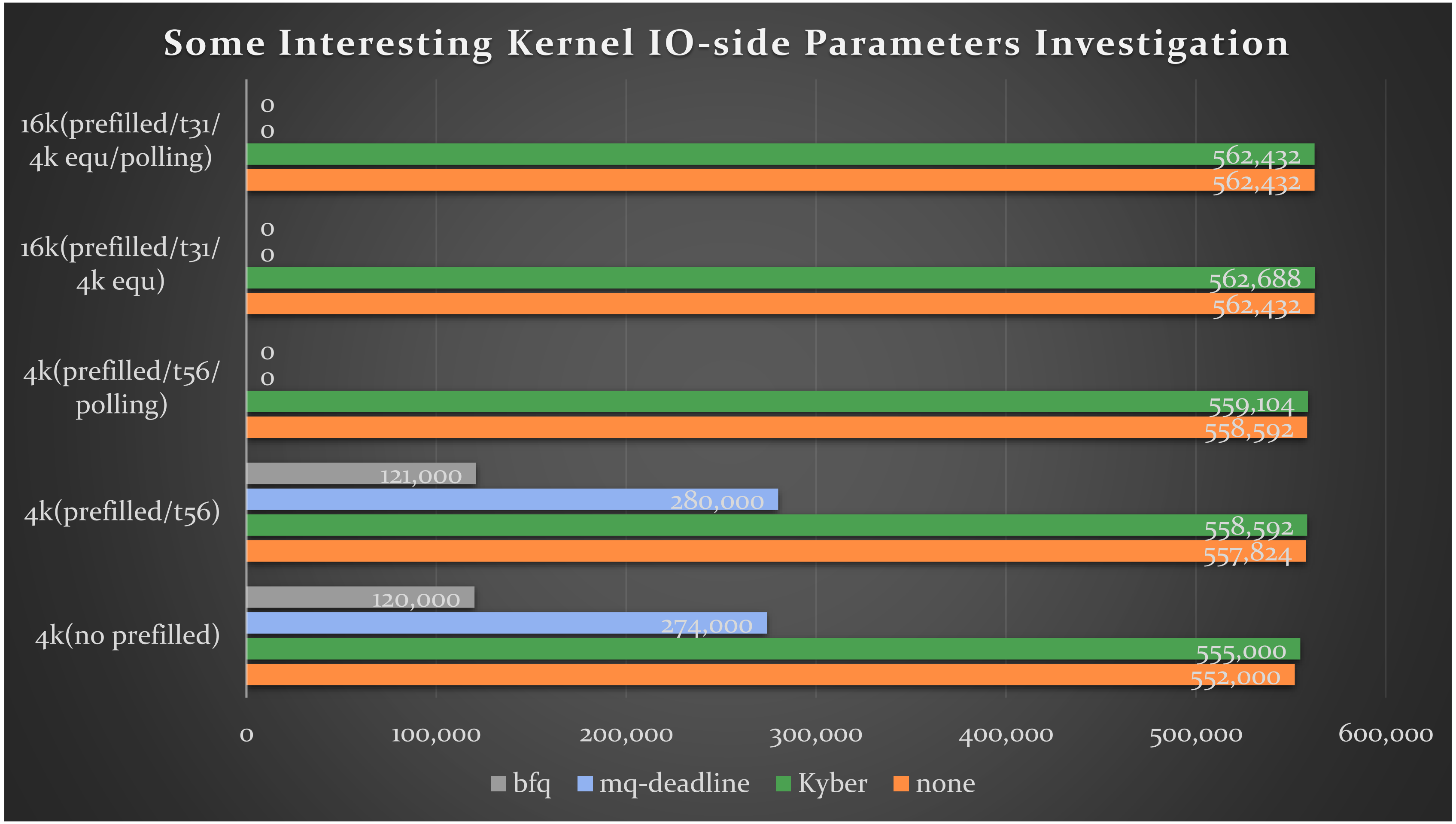
- Adaptive sleep
  - linear backoff algorithm
  - idiomatic rate regulation policy for dismatch-varied P/C systems

- *Conclusion #1*
  - *FS/EXT4 has obvious overheads for fast disks although some FS like XFS may pay more attention on these*
  - *Modern kernel still has many places to improve*
    - *spin lock*
    - *lack resilient configurations*
  - *So, finally, if you need to squeeze out all the performance from fast storages, you may want your own FS or directly work on raw devices*

- Modern features
- *"Have we fully benefited from the dramatically evolving modern features?"*
  - Optane like are game changers (but maybe just for Optane DIMM)
- Software
  - Centos 7.2 is ancient
- Hardware
  - There are pearls out of toothpastes in recent-generation Intel processors

- Investigation#3 (Software) new kernel IO-side parameters
  - RWF_HIPRI(kernel 4.6+)
  - New blk-mq(kernel 3.13+) schedulers
    - *Kyber(Facebook, kernel 4.12+)*

Beyond the Results



Findings:

1. Polling mode slightly positive in 4k but not for >4k
2. Kyber scheduler helps a little in all spectrum

Benchmark method:
running (fio) program interleavingly untill the nexting two runs can not hit higher results, pick up the highest results: none -> kyber -> none -> kyber -> ...

- Investigation#4 (Hardware) Intel TSX-NI
  - Parallel (speculative) execution on multi cores without sync (when no data race)
    - little overhead in the optimistic case
    - This competition just provides a natural optimistic case
    - Broadwell+(GA)
    - Lock-free is hard for complex data structures, and even not necessarily faster than lock
  - NOTE: competition's CPU power overflow, so:
  - *Conclusion#4*
    - *TSX should be first-try alternative for 'old' lock(std::mutex) with basic guards*

- Better designs
  - *"Are the patterns and designs found in the competition valid for the real world engineering?"*
  - Mmap for process crash protection is homework-level simplification
    - Smaller block size(4k) is still best choice for OLTP
  - IO stuffs be tackled at the IO layer
    - sequential loads merged at the IO layer: Kyber like IO scheduler
    - Make your own FS (abcFS … PolarFS)
  - NUMA-ware rather than NUMA-hidden(UMA) – however PolarDB is UMA style…
  - Sync mode is definitely the performance killer, async io is favored in wise practices
    - libaio is truly a good to do async io when play with modern kernel

- System evolve
  - *“Your Part is naïve. So, in practice, we still use B-tree like data structures. Really?”*
  - If distribution of key known well, flat array like indexing (base)structure is enough
  - The Part algorithm works well with recent hot learned index structure* as base structures with extended actions
    - query*, insertion, update(remove) and even concurrent
    - hard part shifted to modelling (via ANN* ...)

* *The Case for Learned Index Structures, 2017-2018, Tim(MIT), Alex, Ed, Jeff Dean(Google) et al.*

The Case for Learned Index Structures

Tim Kraska*
MIT
Cambridge, MA
kraska@mit.edu

Alex Beutel
Google, Inc.
Mountain View, CA
alexbeutel@google.com

Ed H. Chi
Google, Inc.
Mountain View, CA
edchi@google.com

Jeffrey Dean
Google, Inc.
Mountain View, CA
jeff@google.com

Neoklis Polyzotis
Google, Inc.
Mountain View, CA
npolyzotis@google.com

Innovations

- *Proposed patterns to full IO bandwidth*
  - Observations: 562k+iops(w)，605k+iops(r)，42.5M+iops(range)
- *Concise but efficient designs and implementations*
  - Simple sync write taking full use of prebuilt offset (faster than general lock-free queue)
  - O(1) range indexing structure
  - customized built-in replacements are confirmed to be effective for more stable peak (pdbr_cpy/mov/bswap)
  - Tens of designs and options(IO pattern/Comp pattern/core affinity/numa…) have been implemented and compared
- *More exploits to modern hardware*
  - TSX used wisely - Thread safety is guaranteed without performance loss

Innovations

- *Proposed high performance designs works better under more balanced engineering scenarios*
    - ugly tricks that are not engineering discarded
- *Correctness First*
    - Lines of test sources are more than that of implementation sources.
    - Concurrent correctness done even in low contention
- *Measurements based explorations makes optimization scientific*
    - Adaptive sleep
    - TSX optimization
    - Better understandings to kernel and modern engineering

Woooooooooooo

Salute to Aliyun’s Ten years

Thanks for Aliyun, Tianchi, 凝岚

团队成员介绍

金明剑(Jin Mingjian)

Director, Data Department at Tigerjoys

Doctor, University of Chinese Academy of Sciences

- Scala@Google Summer of Code, 2010
- Java8@Landz(battle tested for Netty replacement...), 2013

Interests: Data Engineering with some Data science

Appendix#1

before slow down

Elapsed Time: 10.000s

CPU Time:	2.256s
Instructions Retired:	1,467,839,268
Microarchitecture Usage:	13.7% of
Wait Rate:	8.434
Context Switch Time:	576.964s
Total Thread Count:	58
Paused Time:	0s

Top Hotspots

This section lists the most active functions in your appl performance.

Function	Module	CPU Time
__read_once_size	vmlinux	0.501s
[crc32c_intel]	crc32c_intel	0.100s
jbd2_journal_stop	jbd2	0.088s
queued_read_lock	vmlinux	0.084s
add_transaction_credits	jbd2	0.071s
[Others]		1.412s

after slow down

Elapsed Time: 10.000s

CPU Time:	2.003s
Instructions Retired:	1,241,488,643
Microarchitecture Usage:	14.4% of Pipeli
CPI Rate:	3.326
Wait Rate:	8.306
Context Switch Time:	577.204s
Total Thread Count:	58
Paused Time:	0s

Top Hotspots

This section lists the most active functions in your application. hotspot functions typically results in improving overall applicat

Function	Module	CPU Time
__read_once_size	vmlinux	0.464s
jbd2_journal_stop	jbd2	0.097s
ext4_es_lookup_extent	ext4	0.086s
native_queued_spin_lock_slowpath	vmlinux	0.086s
____versions	jbd2	0.076s
[Others]		1.193s

Appendix#1

before slow down

Function / Call Stack	CPU Time ▼
▼ __read_once_size	0.501s
▼ ↖ native_queued_spin_lock_slowpa	0.501s
▶ ↖ pv_queued_spin_lock_slowpath	0.493s
▶ ↖ pv_queued_spin_lock_slowpath	0.004s
▶ ↖ pv_queued_spin_lock_slowpath	0.004s

after slow down

Function / Call Stack	CPU Time ▼
▼ __read_once_size	0.464s
▼ ↖ native_queued_spin_lock_slowpa	0.464s
▶ ↖ ____versions ← jbd2_journal_g	0.265s
▶ ↖ jbd2_journal_dirty_metadata ←	0.194s
▶ ↖ __jbd2_log_wait_for_space ← j	0.005s

Appendix#1

before slow down

```
vmlinux!__read_once_size - compiler.h
vmlinux!native_queued_spin_lock_slowpath+
vmlinux!native_queued_spin_lock_slowpath+
vmlinux!pv_queued_spin_lock_slowpath+0xa
vmlinux!queued_spin_lock_slowpath - qspinl
vmlinux!queued_spin_lock+0x11 - qspinlock.l
vmlinux!do_raw_spin_lock - spinlock.h:180
vmlinux!__raw_spin_lock+0xa - spinlock_api_
vmlinux!_raw_spin_lock+0x9 - spinlock.c:144
jbd2!____versions+0x1f2f - jbd2.mod.c:27
jbd2!jbd2_journal_get_write_access+0x2c - t
ext4!__ext4_journal_get_write_access+0x2c
ext4!ext4_split_extent_at+0x201 - extents.c:3
ext4!ext4_split_extent+0xc2 - extents.c:3378
ext4!ext4_split_convert_extents+0xac - exter
ext4!ext4_ext_handle_unwritten_extents+0x9
ext4!ext4_ext_map_blocks+0x4ff - extents.c:
ext4!ext4_map_blocks+0xed - inode.c:636
ext4!_ext4_get_block+0x8e - inode.c:785
ext4!ext4_get_block_trans+0x8b - inode.c:84
ext4!ext4_dio_get_block_unwritten_sync+0x3
```

after slow down

```
vmlinux!__read_once_size - compiler.h
vmlinux!native_queued_spin_lock_slowpath+0xcf - qspinlock.c:
vmlinux!native_queued_spin_lock_slowpath+0x2d - qspinlock.c
vmlinux!pv_queued_spin_lock_slowpath+0xa - paravirt.h:679
vmlinux!queued_spin_lock_slowpath - qspinlock.h:32
vmlinux!queued_spin_lock+0x11 - qspinlock.h:88
vmlinux!do_raw_spin_lock - spinlock.h:180
vmlinux!__raw_spin_lock+0xa - spinlock_api_smp.h:143
vmlinux!_raw_spin_lock+0x9 - spinlock.c:144
jbd2!jbd2_journal_dirty_metadata+0x14d - transaction.c:1465
ext4!trace_event_define_fields_ext4_find_delalloc_range+0x65
ext4!ext4_split_extent_at+0x117 - extents.c:3234
ext4!ext4_split_extent+0x10a - extents.c:3339
ext4!ext4_split_convert_extents+0xac - extents.c:3703
ext4!ext4_ext_handle_unwritten_extents+0x9be - extents.c:40
ext4!ext4_ext_map_blocks+0x4ff - extents.c:4345
ext4!ext4_map_blocks+0xed - inode.c:636
ext4!_ext4_get_block+0x8e - inode.c:785
ext4!ext4_get_block_trans+0x8b - inode.c:845
ext4!ext4_dio_get_block_unwritten_sync+0x30 - inode.c:924
vmlinux!get_more_blocks+0xe6 - direct-io.c:711
```

Appendix#2

Top Hotspots

This section lists the most active functions in your application.
performance.

Function	Module	CPU Time
polar_race::pdbr_cpy	gtestAll	0.085s
__mcount_internal	libc-2.28.so	0.035s
polar_race::EngineRace::Write	gtestAll	0.030s
queued_read_lock	vmlinux	0.020s
ext4_mark_iloc_dirty	ext4	0.015s
[Others]		0.301s

Top Hotspots

This section lists the most active functions in your application.
performance.

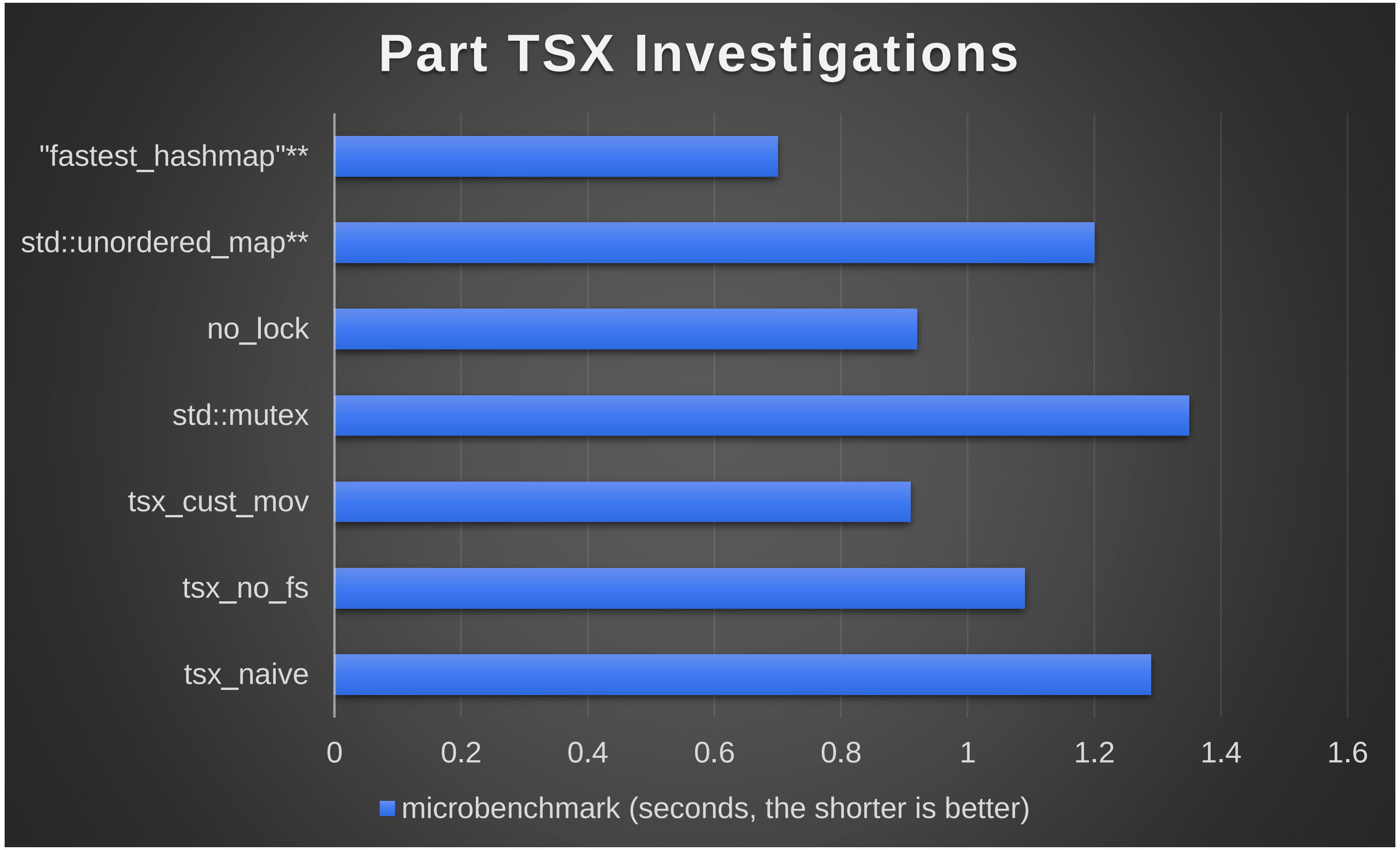
Function	Module	CPU Time
__read_once_size	vmlinux	0.200s
__mcount_internal	libc-2.28.so	0.040s
polar_race::pdbr_cpy	gtestAll	0.025s
ext4_mark_iloc_dirty	ext4	0.020s
[crc32c_intel]	crc32c_intel	0.020s
[Others]		0.271s

Appendix#2

```
vmlinux!__read_once_size - compiler.h
vmlinux!native_queued_spin_lock_slowpath+0xcf
vmlinux!native_queued_spin_lock_slowpath+0x2
vmlinux!pv_queued_spin_lock_slowpath+0xa - pa
vmlinux!queued_spin_lock_slowpath - qspinlock.h
vmlinux!queued_spin_lock+0x45 - qspinlock.h:88
vmlinux!queued_read_lock_slowpath+0x1f - qrwl
jbd2!trace_event_define_fields_jbd2_handle_stat
jbd2!jbd2__journal_start+0xd9 - transaction.c:43
ext4!ext4_dirty_inode+0x2d - inode.c:311
vmlinux!arch_static_branch - jump_label.h:36
vmlinux!static_key_false - jump_label.h:142
vmlinux!trace_writeback_dirty_inode - writeback.h
vmlinux!__mark_inode_dirty+0x41 - fs-writeback.
vmlinux!generic_update_time+0xb6 - inode.c:165
vmlinux!file_update_time+0xe1 - inode.c:1884
vmlinux!__generic_file_write_iter+0x98 - filemap.
ext4!inode_unlock - fs.h:743
ext4!ext4_file_write_iter+0xc6 - file.c:267
vmlinux!new_sync_write+0xfb - read_write.c:476
vmlinux!vfs_write+0x36 - read_write.c:574
```

```
inux/fs.h: * hopefully graduate it to a proper O_CMTIME flag supported by open(2) soon.
inux/fs.h:#define FMODE_NOCMTIME              ((__force fmode_t)0x800)
inux/fs.h:#define S_NOCMTIME    128     /* Do not update file c/mtime */
inux/fs.h:#define IS_NOCMTIME(inode)   ((inode)->i_flags & S_NOCMTIME)
```

Beyond the Results



** **fastest_hashmap** and **undereded_map** are single thread 1M k/v insertions(and **no_lock** are not thread safe structure), just added for infos

fastest_hashmap got from https://probablydance.com/2017/02/26/i-wrote-the-fastest-hashtable/

* microbenchmark infos: 52 threads, 1M K/V insertions per thread, 32k slots per Part, 2048 Parts; pick best in five runs

Appendix#3

Part TSX Optimizaition Steps

tsx_naive → tsx_no_fs → tsx_cust_mov

	tsx_naive	tsx_no_fs	tsx_cust_mov
Clockticks:	116,861,548,459	93,859,249,077	82,132,208,750
Transactional Cycles:	51,544,275,876	46,667,331,412	42,115,776,490
Abort Cycles:	14,790,221,004	7,005,122,870	367,151,189
Abort Cycles (%):	28.694	15.011	0.872
Total Thread Count:	53	53	53
Paused Time:	0s	0s	0s

	tsx_naive	tsx_no_fs	tsx_cust_mov
TSX Aborts:	**13,920,000**	**5,090,000**	**140,000**
Instruction:	4,140,000	4,290,000	0
Data Conflict:	9,750,000	790,000	140,000
Capacity:	10,000	0	0
Other:	20,000	10,000	0
Total Thread Count:	53	53	53
Paused Time:	0s	0s	0s