kaushik, Pradeep, kan

McKusick, M.K., Joy, W.N., Leffler, S.J., and Fabry, R.S.
**A Fast File System for UNIX**
ACM Transactions on Computer Systems, Vol. 2, No. 3, August 1984, pp. 181-197.

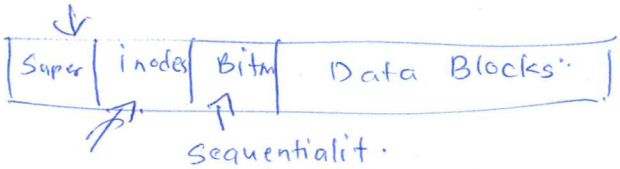
(1) Motivation: Improve throughput of I/O, fragmentation, random IOs.

(2) Most files are small. ~~Access patterns follow locality.~~ ~~local accesses are~~ local ~~work~~ and sequential accesses are much faster than random accesses.

Previous work uses the free list for allocation, which lead to fragmentation.

(2) Main Contribution: Makes the block size larger. Also efficiently handles small files with internal fragmentation; Improve locality by introducing group cylinder; Bitmap rename, symbolic, longer filenames, Quotas.

(3) Cylindrical Groups.




Allocation of cylindrical groups
- same file in a group
- same dir in same group.

(X)

Big Block size can be divided internally.

Additional functional calls for perf. & usability;

---

Main contribution → - Having local information clustered together (eg) inodes & data belonging to files within same directory are kept in same cylinder groups. Better Locality.
- Coalescing of small fragments into blocks - Flexibility of dynamic file sizes.

## Evaluation

Inode accesses went down because of inode grouping together on disk for the same directory.

unsolved problem: Random I/O had a lot of overhead if you had to access different directories, spread out files.

Tyler Harter, Chris Dragga, Michael Vaughn, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau

**A file is not a file: understanding the I/O behavior of Apple desktop applications**

SOSP '11 Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles

Group 9:

Problem/Motivation:-

→ File Applications access files using multiple threads, use reckless synchronization; lack sequential access. Frameworks introduce file access overheads ~~even~~ without the developer realizing it - Analyze workload I/O patterns of desktop users

Contribution: iBench Suite which runs a variety of common applications on a typical home workstation and study the I/O access patterns. Main observations were many auxiliary files, multiple threads perform I/O, a file can have a complicated underlying structure etc.

Details

Many rename() and fsync() operations. Figure 4 shows that most file accesses are for very small (< 4Kb) files, Figure 12 (fsync) & 14 (rename) show these characteristics

- Single file is not a file but a filesystem.

(4) More Details.

- Sequential is not sequential.
- Auxilary files dominates
- Frameworks influence I/O.
- Preallocation and hints are not that frequently used in consumer level application.

(See back side of paper)

- fsync is called frequently for data persistence.

Future direction:

How to make application that works better with FS; ~~Or FS that~~ be

Patterson, D., Gibson, G., and Katz, R.,
**A Case for Redundant Arrays of Inexpensive Disks (RAID)**
Proceedings of the 1988 ACM SIGMOD Conference on Management of Data, Chicago IL, June 1988.

Motivation

(& memory speeds)

Disk performance has not kept up with Moore's Law. Need to increase disk throughput. Single disks are susceptible to failure. High performance disks are expensive. Disks have become bottleneck in applications. Need to increase reliability (failures). more string

(2) Way to organize inexpensive disk to achieve better performance & reliability compared to a single expensive disk.

→ Detailed performance analysis for all RAID systems & applications suited for each level.

(3) Tech details:

RAID-0: Striping

| D0 | D1 | D2 | D3 |
|---|---|---|---|
| 0 | 1 | 2 | 3 |
| 4 | 5 | 6 | 7 |

RAID-1: Mirroring

| D0 | D1 | D2 | D3 |
|---|---|---|---|
| 0 | 0 | 1 | 1 |
| 2 | 2 | 3 | 3 |

RAID-4: Parity

| D0 | D1 | D2 | D3 | P4 |
|---|---|---|---|---|
| 0 | 1 | 2 | 3 | P0 |
| 4 | 5 | 6 | 7 | P1 |

RAID-5: Rotated Parity:

| D0 | D1 | D2 | D3 | P4 |
|---|---|---|---|---|
| 0 | 1 | 2 | 3 | P0 |
| 5 | 6 | 7 | P1 | 4 |
| 10 | 11 | P2 | 8 | 9 |
| 15 | P3 | 12 | 13 | 14 |
| P4 | 16 | 17 | 18 | 19 |

(4) They defined MTTF to compare different RAID ~~system~~ configurations

$$MTTF_{RAID} = \frac{(MTTF_{Disk})^2}{(D + C * n_g) * (G + C - 1) * MTTR}$$

Evaluation: Raid 5 is better than 4 always

Unsolved Problems → 12 problems discussed at the end of the paper. Such as -

1. Effect of RAID on latency
2. Impact on MTTF calculation of non-exponential failure assumption.

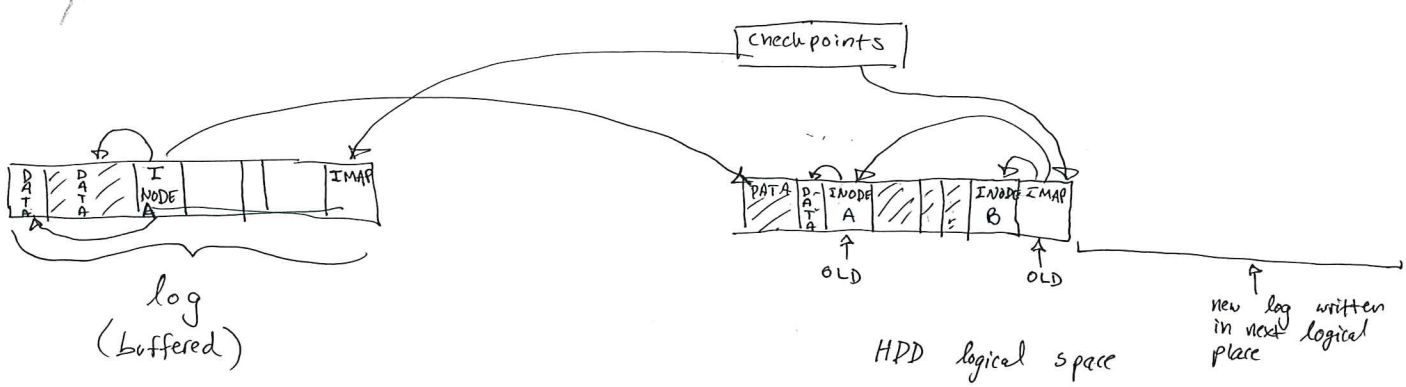
Rosenblum, M. and Ousterhout, J.
**The Design and Implementation of a Log-Structured File System**
ACM Transactions on Computer Systems, Vol. 10, No. 1, February 1992, pp. 26-52.

⇒ Earlier systems were not leveraging. (1)

Contribution: (1) Improve the disk usage from 5-10% to 70%. by writing sequentially.

(2) Cleaning policy → Cold segment free space was much more valuable; so they cleaned cold segment at 15% while hot segments at a much higher utilization ratio.

3.)




* Checkpoint region in fixed location points to IMAPs

main contribution: Indirection → faster 'segsequential' random I/O

4) segment usage table → to help cleaner treat hot and cold segments differently. Pick segment with maximum cost-benefit ratio.

5) Directory operation log → to ensure consistency between directory & inodes during roll forward.

Sprite LFS - performed better for all workloads other than sequential ~~overwrite~~ reread.

Future work

Support multiple size segments.

How do you optimize LFS for SSDs?

**The Unwritten Contract of Solid State Drives**

Jun He, Sudarsun Kannan, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau

Proceedings of the 20th European Conference on Computer Systems (EuroSys '17)

(1) There was no prior work for formalizing good access patterns for SSDs. There was such formalizing present for HDDs, and many existing file systems were optimized for this. This work tries to formalize good access rules for SSDs. They conduct a vertical analysis of applications, file systems & SSD. Prior work was distributed across different systems for SSD optimized designs

(2) Unwritten Contract → tool to analyze the contracts

Bottlenecks for current application w.r.t. SSDs.

+ discovering the unwritten contract for designing high performance applications and file systems

(3) +wiscsee; wiscsim + w.r.t (compare) different file-system

Technical contributions / findss

Metrics for SSD operations = NCQ depth, zombie curves

(4) The 5 rules: request scale, locality, align sequentiality, grouping by death time, uniform data lifetime

Other performance metrics: request size, miss ratio curve, unaligned ratios

Evaluation: F2FS suffers from a lot of overheads including the fact that its garbage collection interferes with the SSDs garbage collection.

Future work

1. Trade off b/w different rule for cont (Performance) optimization.
2. Validity of NisCSim with actual SSD performance

Minh - Suchita - Song

Aapo Kyrola and Guy Blelloch and Carlos Guestrin
GraphChi: Large-Scale Graph Computation on Just a PC.
USENIX Symposium on Operating Systems Design and Implementation (OSDI'12).

* Problem
  * large graph computation is a challenge with computational resources, especially on distributed systems.

=> Goal: execute several advanced data mining, graph mining, and machine learning on large graph, using just a consumer-level computer; also supports graphs that evolve over time.

Why Previous Work not sufficient:-
- did not allow changes to graph, designed for graph traversals not computation

Main Contribution: GraphChi used pre-processing of the graph to allow the parallel sliding window algo sequentially read the graph from disk. This locality of edges and vertices in each sliding window is what makes this efficient.

Kevin - Peter - Abhinav

Details: Parallel Sliding Window -- divide the graph; Sort the edges acc. to source Id & group according to destination vertex. When traversing each shard had contiguous area in other shards to make access efficient.

→ Edge buffers for evolving graphs

Technical Contributions

→ They analyze how GraphChi works with evolving graphs. They push GraphChi to its limits w.r.t to adding edges. (Figure 9)

→ They compare GraphChi with existing distributed platforms for the same work. algorithm. (Table 2)

5. Evaluation:

Table 2: Hadoop performances poorly. As there is no nice way to split. And there were lot of message passing.

Fig 8 c) No. of shard did not vary the performance much.

Evolving Graphs: Could adopt to high rates.

6. Future work: Can we reduce the overhead of pre-processing (sort) in GraphChi?

Amitabha Roy, Ivo Mihailovic, Willy Zwaenepoel
**Xstream: Edge-centric graph processing using streaming partitions**
Symposium on Operating Systems Principles (2013).

1) Problem (+)

To perform graph computation on a single machine.

Motivation: sequential access is much faster than random access

Previous works: 1) vertex - centric 3) huge preprocessing cost (sorting)
2) stream unordered list

~~They also do some pre-processing (like sorting) for to improve performance.~~

(2) Main Contribution.
+ Introduce edge-centric processing as a new model
+ Using streaming partitions both for in-memory and out-of-core graphs
+ Provide Avoid preprocessing

(3) details.

(1) edge centric approach, no pre-processing required.
map-reduce-like execution model. the exec is divided. Scatter - sorting (in memory) - gather.
shuffling.

(2) Generic framework to transit b/w in-memory & out-of-core graphs.

Having too many or too few partitions increases the processing time of a graph significantly. Need to find a happy medium.
They showed resort in GraphChi causes additional, unmentioned work.

Question:-

Does X-Stream scale to more than 1 disks?

Da Zheng and Disa Mhembere and Randal Burns and Joshua Vogelstein and Carey E. Priebe and Alexander S. Szalay
**FlashGraph: Processing Billion-Node Graphs on an Array of Commodity SSDs.**
Conference on File and Storage Technologies (FAST 2015)

1.) Motivation: Utilize SSD for random I/O; Previous works didn't use random I/O and assumptions were made that other processing mechanisms were geared towards HDDs. X-Stream & graphchi had a runtime that was proportional with iterations whereas FlashGraph was more selective in its accesses.

2) Contribution: FlashGraph uses the idea of splitting vertices and edge lists between main memory and disk. The edge lists are up to 35 times larger than vertex sets and FlashGraph lets application scale beyond main-memory.

(3) ~~Just-in-time~~ vertex scheduler, reduces total I/O as sequential vs random performance gap is less in SSD. SAFS file system, merge I/Os.

(4) More tech details: Save space by choosing to compute some vertex info. at runtime, such as the location of an edge list on SSDs and vertex ID; It hides explicit synchronization from users, and bundle messages in a single packet to reduce synch overhead.

(5) Flashgraph vs in-memory engines.
Similar or better than powergraph and Galois (state-of-the art in-memory systems) except for in graph traversal ~~edge~~ application such as BFS → as it would require large amount of data reads from SSD.

(6) Flashgraph vs External memory engines
Outperforms GraphChi & X-Stream by 2 order of magnitude. Has smaller or comparable memory usage compared to the two.

(6) Unsolved problem: ① They assume fixed number of running values in a Thread but dont tackle the case of working Graphs.

② If there are too many updates in graph leading to large message queues, then it wouldnt fit in the cache. This could be a problem in Flash Graph implementation.

Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau

**WiscKey: Separating Keys from Values in SSD-conscious Storage**

Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST '16)

Goal.: Modify Level DB to get more performance out of S.S.D. Reducing Read & Write Amplification. ✓

Current System not adequate because gap b/w Random & Sequential IO bandwidth is closing.

Main Contribution

Separate keys & values to make LSMT smaller for faster decreased read & write amplifications.

Details: → Smaller LSMT reduces write amplication during merging levels

→ Also reduces read amplification as lesser levels are queried during lookup.

→ ~~Range~~ For range query performance, ~~use~~ leverage inherent parallelism of SSDs.

→ Back pointer i.e. store keys with values in the log to improve ~~gg~~ garbage collection performance.

Evaluation: The sequential, random workload performance is much better in ~~Wisc~~ WiscKey than LevelDB. LevelDB has higher CPU usage for seq. workload. For other workloads, WiscKey has higher CPU usage.

6. Future work:

- Can we extend this to other LSMT d b's.
- Apply the same idea for other applications.

**All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications**

Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau

Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI '14)

1. Problem / motivation / goals +
   1. what are the behaviors exhibited by modern file systems that are relevant to building crash-consistent applications?
   2. Do modern applications implement crash consistency protocols correctly?

2. Contribution
   1. Developed Bob to ~~find~~ study Linux file systems in various configurations
   2. Developed Alice to analyse application-level crash consistency.

Details ⇒ Breakdown of operations into micro operations and use of the ~~affected workloads~~ for the given workload, the application assumptions model that of the stresses of the operations done by the File System.

Create a framework which took in APM (File System) and application snapshot with a checker to automatically detect problems in a general way.
Studied 11 applications and found 60 vulnerabilities.

They propose how new file systems can be evaluated using ALICE & BOB, to find vulnerabilities. (Section 4.6)

Group 9

Vijay Chidambaram, Thanumalayan Sanakaranarayana Pillai, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau
**Optimistic Crash Consistency**
Symposium on Operating System Principles, SOSP 2013

Group 9 1) Solves problems of using too many flushes in journal to ensure consistency. current technique is too pessimistic.

Group 4 2) Contribution: Separated ordering and durability and thus reduced the number of flushes to disk required for applications that do not have stringent durability requirements. Implement a ~~probabilistic journaling~~ optimistic journaling approach with linux ext4 variant and called it OptFS. "Asynchronous Durability Notifications", Use of checksums with data, osync & dsync, other techniques like in order journal recovery, delayed writes, etc.

3) • Detailed analysis of probabilistic crash consistency. In particular, they analyze which type of orderings occur in different applications: Early checkpoint, early commit, transaction misorder, mixed. They conclude that early commit is a major contributor to probability of inconsistency. (Figure 3 in paper)
• They analyze the affect of queue size and journal distance on probability of inconsistency.
• They came up with technique to either not using flush or using it later to make systems
(+) faster but consistent
◦ They develop OptFS → remove flushes. D, JM, JC are done ~~first~~ and wait for ADNs Before doing Checkpointing
→ two new system calls for durability (dsync) & ordering (osync), rather than a single heavy duty fsync.

Evaluation → - Reliability was quite high for OptFS even for multiple block changes.

- Due to selective data journaling, it causes two data writes for every block, one to the journal & one to the data block. Performance is bad in this case.

Future Work

Regarding point (2) in Evaluation, an alternative to selective data journaling that would provide both consistency and performance.

Howard, J.H., Kazar, M.L., Menees, S.G., Nichols, D.A., Satyanarayanan, M., Sidebotham, R.N., and West, M.J.
**Scale and Performance in a Distributed File System**
ACM Transactions on Computer Systems, Vol. 6, No. 1, February 1988, pp. 51-81.

Problems they're solving: ① Scalability — NFS is not scalable beyond a small number of machines

(1)

② Granularity is increased from block to file so it needs not require as much dealing with issues like consistency on at block level.

Assumption: Very few users working on same file at a time

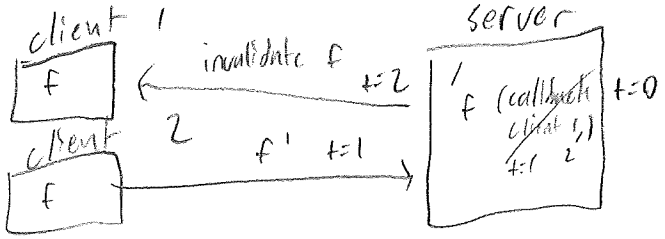
Previous: No transparent file locality in NFS. Because NFS does not work at the granularity of file level, consistency overhead is higher than AFS.

Main Contributions: ① Add state to the servers to improve scalability;

② whole file caching reduced network I/O

Technical Contribution:

Reducing server load by whole-file caching on clients, and using callbacks to avoid unnecessary network requests to the server using additional server state. Consistency with minimal overhead.



## Main contribution

~~Since multiple threads~~ Context switching between light weight processes is less of an overhead. They use light weight processes in the server and one client uses only one of these inside the server.

## Future work

Supporting disconnected clients :)

Efficient replication of volumes, consistency guarantees with data.

Alternatives to whole file caching.

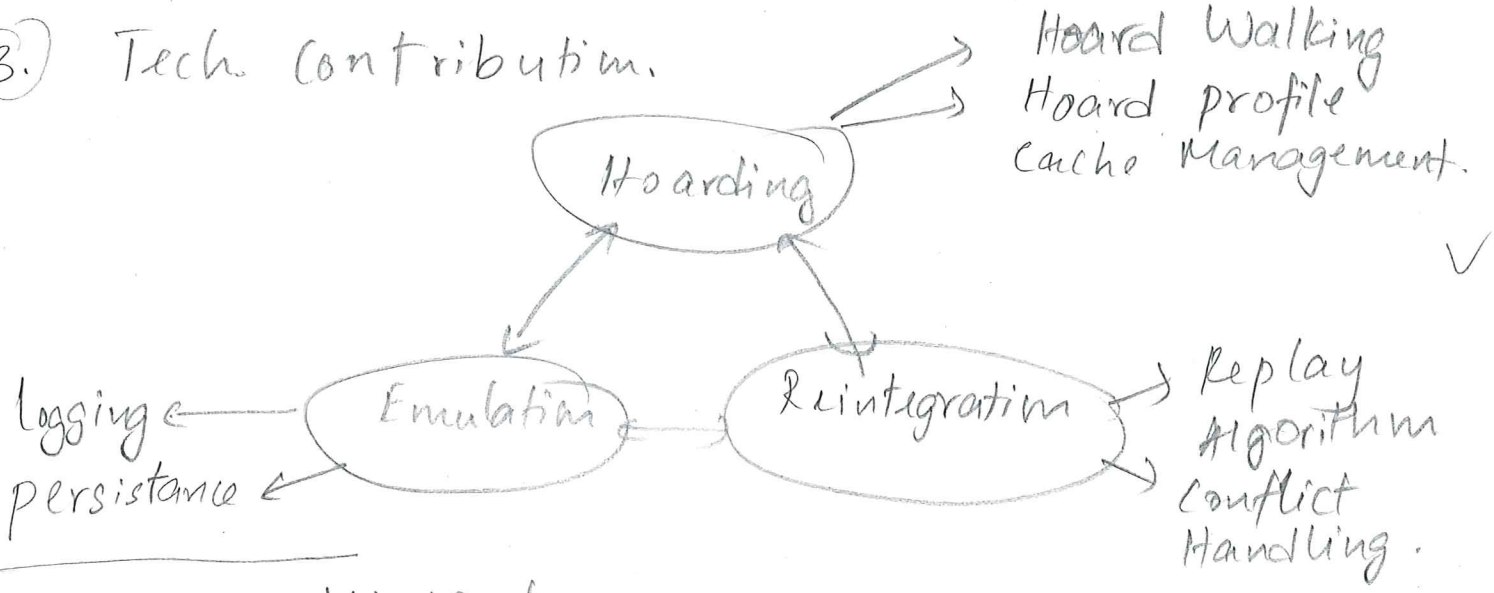
Kaushik, Pradeep, Ram

James J. Kistler and M. Satyanarayanan
Disconnected operation in the Coda File System
ACM Transactions on Computer Systems (TOCS) 10(1), February 1992, 3-25.

① Motivation: Ability to continue operation when the local system is disconnected from the network.
Assumption: Large number of clients with comparitively unreliable local storage. Conflicting accesses are rare.
Previous work like AFS focussed on using caching for performance, whereas CODA uses caching for availability. & performance

② contribution: achieve disconnected operation through optimistic replica control.
It handles both voluntary and involuntary disconnection situations.

③ Tech. Contribution.




* Coda MiniCache...

as of caching strategy, coda supports profile-based prefetching to ensure caching files likely to access in the future. (Hoard Walking)

Evaluation: - Validated their assumption about same file conflicts by observing 0.75% chance of diff users modifying same object less than a day part.

✓

(6.) They make the assumption of very low contention on files. More work should be done on how to resolve conflicts intelligently.

C.A.R. Hoare
**Monitors: An Operating System Structuring Concept**
Communications of the ACM 17, 10, October 1974, pp. 549-557

1. Motivation: Explore monitors as a method to structuring OS.
Goal: Synchronization primitive provides ways to access shared resource.
Assumptions: ~~Provide semantics~~. Signal would be immediately followed by the waiting process.

Current Teching - semaphores, requires internal state tracking, not efficient.

Main contribution - Object oriented structure of managing synchronization of shared data/variables.
Allows for providing guarantee of existing process being run just after signalling process.

The idea of monitor is only 1 process at a time been executed in monitor, automatically acquires lock on entry and release on exit

Mesa has 3 type of procedures ① ~~entry~~ acquire lock ② already has lock ③ external ~~three~~

The semantics of Hoare's monitors use signals to establish invariants on the waiting processes. He shows how monitors and this approach to signals is flexible yet powerful by giving example implementations for various resource monitors.

Evaluation. Hoare's paper is mostly about establishing the concepts of monitors. They haven't given too much details about the evaluation.

Minh- Suchita- Song

Butler W. Lampson, David D. Redell
**Experiences with Processes and Monitors in Mesa**
Communications of the ACM, 23 2, February 1980, pp. 105-117.

Problems: previous problems with monitors (..)
Previous work did not deal with many of the practical considerations of monitors, talked about them in the abstract.

Main motivation: Provide concurrent accessible data with contention minimized. Provided ability for timeouts, aborts, and broadcasts. Invariant was set such that locks are taken upon entry of a monitor and released upon exit.

Contribution - Make an actual working system to provide concurrency semantics.

Introduced finer grained (more specific) concurrency mechanisms like timeout, abort, broadcast, notify that allowed for applications to make better use of concurrency data structures.

5. Evaluation:
Evaluated on 3 applications.

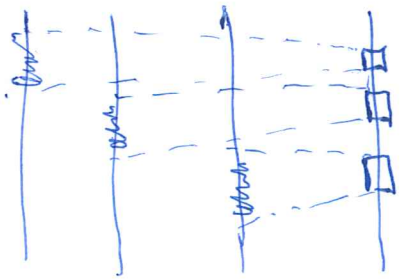
Unsolved Problem: 1. Priority inversion is mentioned but no effective solution is proposed.

Jean-Pierre Lozi and Florian David and Gael Thomas and Julia Lawall and Gilles Muller, Remote Core Locking: Migrating Critical-Section Execution to Improve the Performance of Multithreaded Applications, USENIX Annual Technical Conference (ATC'12), 2012.

1.) Motivations: locks don't scale and contention arises with the rising number of cores additionally, cache synchronization has high overhead when multiple cores are trying to go into a critical section. Previous works still suffered overheads from server / client delegation and roles.

2) Delegating the handling of critical section to a dedicated server using RPC. A profiling tool analyses the various locks in an application and converts the highly contended lock to a dedicated server.

3)




Execute critical section on server core, using a "mailbox" approach. Two advantages:

① Fewer cache misses as the common code is executed on the same ~~thread~~ core, and shares the cache.

② ~~Reduced lock contention~~ No communication overhead of transferring critical data to different cores.

4) Using profiling to determine when to use RCL vs. when to use regular POSIX lock.

(time spent proportion) >20% >70%

2 Metrics 1: use critical section time as indicator of contention.
2: cache locality

3 For Memcached SET. even though RCL scales much better than other locks, but it's still not linear. RCL did perform worse than other lock when use 4 core or less, but ~~out~~ out scale others due to better cache locality.

For BD. RCL has more through put than other locks.

Future work

- more evaluation on multi-server case for handling different critical sections.
- didn't talk about nested locking in detail

Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek,
Robert Morris, and Nickolai Zeldovich

**An Analysis of Linux Scalability to Many Cores**

9th Symposium on Operating Systems Design and Implementation (OSDI), October 2010

Goal: To show that current O.S. (linux kernel) (& application code) can be modified to get good scalability.

Related work: Previous studies have contributed to scalability improvements; but no recent work has looked at performance. Not sufficient → Wide variety of problems, so not enough applications to stress each O.S. component.

Assumption: in-memory file system.

Main Contribution

Patching the linux kernel to achieve better scalability (more linear) with increasing number of CPU cores. Introducing/adding data structures (sloppy counters, per core DSes, lock free structures) that enable greater scalability. ✓

Details :-

TempFS is used to eliminate bottlenecks due to disk. They test wide variety of applications such as Mail Server, Databases, Object caches and web servers.

Evaluation :

Out of the seven application included in MOSBENCH, only gmake can scale without any modification, because most of time it spent in CPU is on compiler, which can run independently.

All applications in MOSBENCH are limited by either hardware (NIC, cache ...) or application structure (spin lock ...), but not linux itself

6. Future Work:

* For hardware bottlenecks, should we revisit once it is not a bottleneck.
* Can we provide stronger guarantees for the linux kernel design.

Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert T. Morris, and Eddie Kohler
**The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors.**
24th ACM Symposium on Operating Systems Principles (SOSP), November 2013.

1. Motivation: In past, we try to scale system after we build it. Is there a way to design interface which is scalable.
Goal: Provide sufficient condition for scalable API. Build tools to verify it.
Assumption: Past method we're too late in the dev. phase.

---

Contribution
(+1) Existing applications can run on multi-core processor without rewriting the whole application. By analysing an application interface, software commute generate list of test cases which tells all conflict operation even before implementing.
+ They defined scalable commutativity rule Minh-Suchita-Scop

(+1) Technical contribution or finding
+ Whenever interface operations commute, they can be implemented in a way that scales
+ The rules to design commutative interfaces (decompose compound operations, non-determinism specification, weak ordering, release resource asynchronously)

- Commutativity leads to cache like (unchurning) ??

Evaluation . Fig 7.

The usage of commutative APIs doesn't always result in linear scalability with the no. of cores.