REMIX: Efficient Range Query for LSM-trees

Wenshao Zhong, Chen Chen, Xingbo Wu, Song Jiang; University of Illinois at Chicago, University of Texas at Arlington

- LSM-tree-based KV stores organize data in a multi-level structure for high-speed writes while their **range queries are less efficient and costly**.
  - **Seek** and **sort-merge** will be invoked involving multiple table files but current structure does not consider much about it.

1. LSM-tree's **out of place scheme** comes with penalties on search efficiency**.** As keys in a range may reside in different tables, potentially slowing down queries because of high computation (*pull index to memory*) and I/O costs (*wasted I/O*); In a word, **Less time blocks retrieving vs. Compaction burden**.
2. **Current optimizations are less efficient:** *in-memory bloom filter; range filter.*
3. **Key Insight: KV-pairs do not have to be physically sorted, and keep its data logically sorted is a better choice because of better data locality and less data re-arrangement:**
   - **sorted-views**, constructed by **A** range query on multiple SSTables, are repeatedly reconstructed at search time and immediately discarded afterward, which leads to poor search performance and wasted I/O and computation.
   - *Global-sorted indexing for each key is efficient but must take massive space.* ***Maybe reusing the sorted-views.***

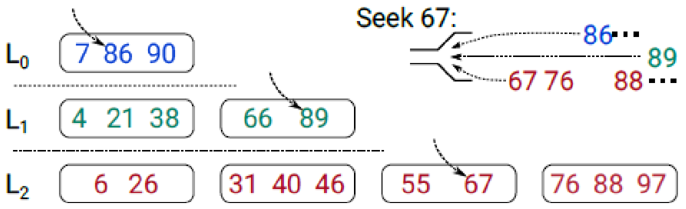

Figure 1: An LSM-tree using leveled compaction

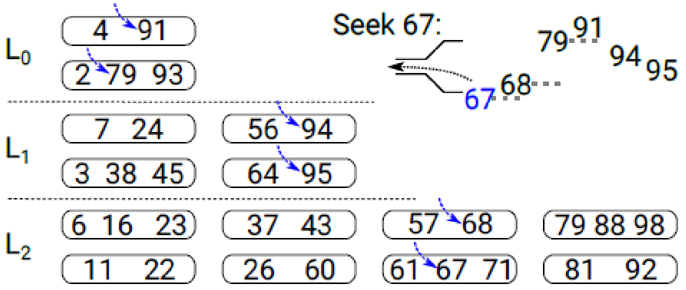

Figure 2: An LSM-tree using tiered compaction

REMIX: Efficient Range Query for LSM-trees

Wenshao Zhong, Chen Chen, Xingbo Wu, Song Jiang; University of Illinois at Chicago, Univer

- LSM-tree-based KV stores organize data in a multi-level structure for high-speed w ... s **efficient and costly**.
  - **Seek** and **sort-merge** will be invoked involving multiple table files but current structure

1. LSM-tree's **out of place scheme** comes with penalties on search efficiency. As key... tables, potentially slowing down queries because of high computation (*pull index to memory*) and I/O costs (*waisted I/O*); In a word, **Less time blocks retrieving vs. Compaction burden**.
2. **Current optimizations are less efficient:** *in-memory bloom filter; range filter.*
3. **Key Insight: KV-pairs do not have to be physically sorted, and keep its data logically sorted is a better choice because of better data locality and less data re-arrangement:**
   - **sorted-views**, constructed by **A** range query on multiple SSTables, are repeatedly reconstructed at search time and immediately discarded afterward, which leads to poor search performance and wasted I/O and computation.
   - *Global-sorted indexing for each key is efficient, but must take massive space.* ***Maybe reusing the sorted-views.***

REMIX: Efficient Range Query for LSM-trees

Wenshao Zhong, Chen Chen, Xingbo Wu, Song Jiang; University of Illinois at Chicago, University of Texas at Arlington

- Propose REMIX (**R**ange-query **E**fficient **M**ulti-table **I**nde**X**), without struggling between physically rewriting data and performing expensive sort-merging. In a word, **reusing sorted view of range query.**
  - REMIX employs a space-efficient data structure to record a globally sorted view of KV data spanning multiple table files. It can take advantage of a write-efficient compaction strategy without sacrificing search performance.
- **REMIX Data structure**: maintaining keys with a globally-sorted manner with granularity of segments.
  - Divide the keys of a **sorted view** into segments, each containing a fixed number of keys; each segment is attached with an *anchor key*, a set of *cursor offsets*, and a set of *run selectors;* conduct **binary search** for segment **outside & inside** searching.
- **RemixDB**: divides the key space into partitions of non-overlapping key ranges. The table files in each partition are indexed by a **REMIX**, providing a sorted view of the partition. It is a ***single-level LSM-tree using tiered compaction***.
  - A compaction in a partition creates a new version of the partition that includes a mix of new and old table files and a new REMIX file. The old version is garbage-collected after the compaction.
    - **Minor compaction**: writes KV from the ImmuTable into a partition without rewriting existing table files, but rebuilds REMIX of the partition.
    - **Major compaction:** is required when exceeding partition's threshold T. A major compaction sort-merges existing table files into fewer ones.
    - **Split Compaction**: tables in each partition can be reduced.

REMIX: Efficient Range Query for LSM-trees

Wenshao Zhong, Chen Chen, Xingbo Wu, Song Jiang; University of Illinois at Chicago, U

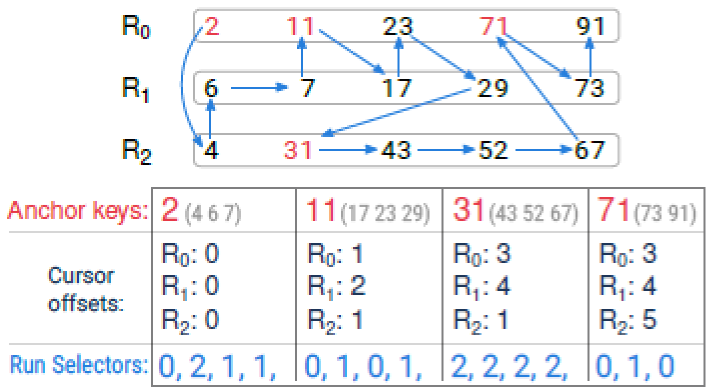

Figure 3: A sorted view of three sorted runs with REMIX

- Propose REMIX (**R**ange-query **E**fficient **M**ulti-table **I**nde**X**), without strugglin performing expensive sort-merging. In a word, **reusing sorted view of range q**
  - REMIX employs a space-efficient data structure to record a globally sorted view of KV data spanning multiple table files. It can take advantage of a write-efficient compaction strategy without sacrificing search performance.
- **REMIX Data structure**: maintaining keys with a globally-sorted manner with granularity of segments.
  - Divide the keys of a **sorted view** into segments, each containing a fixed number of keys; each segment is attached with an *anchor key*, a set of *cursor offsets*, and a set of *run selectors;* conduct **binary search** for segment **outside & inside** searching.
- **RemixDB**: divides the key space into partitions of non-overlapping key ranges. The table files in each partition are indexed by a **REMIX**, providing a sorted view of the partition. It is a ***single-level LSM-tree using tiered compaction***.
  - A compaction in a partition creates a new version of the partition that includes a mix of new and old table files and a new REMIX file. The old version is garbage-collected after the compaction.
    - **Minor compaction**: writes KV from the ImmuTable into a partition without rewriting existing table files, but rebuilds REMIX of the partition.
    - **Major compaction:** is required when exceeding partition's threshold T. A major compaction sort-merges existing table files into fewer ones.
    - **Split Compaction**: tables in each partition can be reduced.

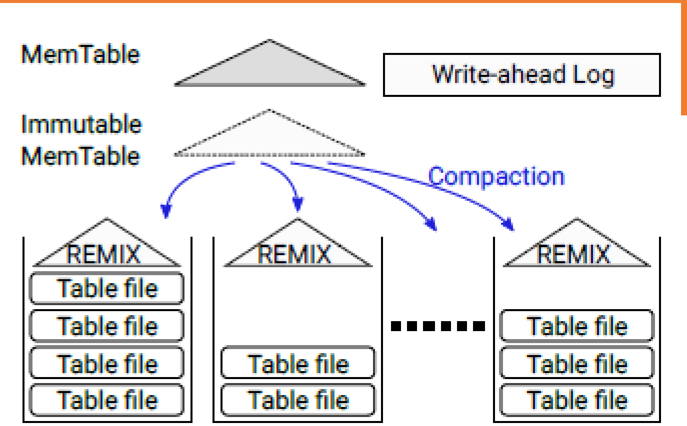

Figure 5: Overview of RemixDB

REMIX: Efficient Range Query for LSM-trees

Wenshao Zhong, Chen Chen, Xingbo Wu, Song Jiang; University of Illinois at Chicago, Univer

- Propose REMIX (**R**ange-query **E**fficient **M**ulti-table **I**nde**X**), without struggling bet performing expensive sort-merging. In a word, **reusing sorted view of range quer**
  - REMIX employs a space-efficient data structure to record a globally sorted view of KV data spanning multiple table files. It can take advantage of a write-efficient compaction strategy without sacrificing search performance.
- **REMIX Data structure**: maintaining keys with a globally-sorted manner with granularity of segments.
  - Divide the keys of a **sorted view** into segments, each containing a fixed number of keys; each segment is attached with an *anchor key*, a set of *cursor offsets*, and a set of *run selectors;* conduct **binary search** for segment **outside & inside** searching.
- **RemixDB**: divides the key space into partitions of non-overlapping key ranges. The table files in each partition are indexed by a **REMIX**, providing a sorted view of the partition. It is a ***single-level LSM-tree using tiered compaction***.
  - A compaction in a partition creates a new version of the partition that includes a mix of new and old table files and a new REMIX file. The old version is garbage-collected after the compaction.
    - **Minor compaction**: writes KV from the ImmuTable into a partition without rewriting existing table files, but rebuilds REMIX of the partition.
    - **Major compaction:** is required when exceeding partition’s threshold T. A major compaction sort-merges existing table files into fewer ones.
    - **Split Compaction**: tables in each partition can be reduced.

REMIX: Efficient Range Query for LSM-trees

Wenshao Zhong, Chen Chen, Xingbo Wu, Song Jiang; University of Illinois at Chicago, Uni

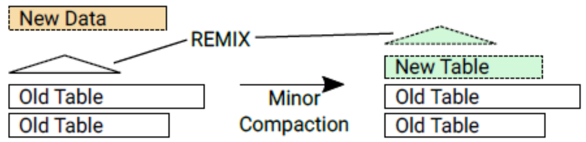

Figure 8: Minor compaction

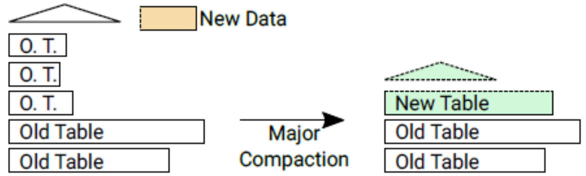

Figure 9: Major compaction

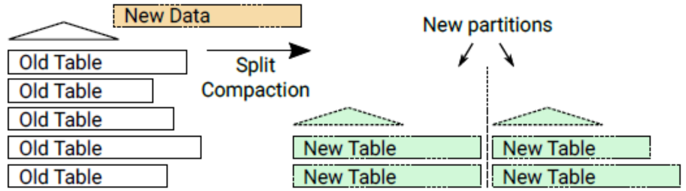

Figure 10: Split compaction

- Propose REMIX (**R**ange-query **E**fficient **M**ulti-table **I**nde**X**), without struggling d performing expensive sort-merging.
  - REMIX employs a space-efficient data structure to record a globally sorted view of can take advantage of a write-efficient compaction strategy without sacrificing search pe
- **REMIX Data structure**: maintaining keys with a globally-sorted manner with g
  - Divide the keys of a sorted view into segments, each containing a fixed number of keys; each segment is attached with an *anchor key*, a set of *cursor offsets*, and a set of *run selectors;* conduct **binary search** for segment **outside & inside** searching.
- **RemixDB**: divides the key space into partitions of non-overlapping key ranges. The table files in each partition are indexed by a **REMIX**, providing a sorted view of the partition. It is a ***single-level LSM-tree using tiered compaction***.
  - A compaction in a partition creates a new version of the partition that includes a mix of new and old table files and a new REMIX file. The old version is garbage-collected after the compaction.
    - **Minor compaction**: writes KV from the ImmuTable into a partition without rewriting existing table files, but rebuilds REMIX of the partition.
    - **Major compaction:** is required when exceeding partition's threshold T. A major compaction sort-merges existing table files into fewer ones.
    - **Split Compaction**: tables in each partition can be reduced.

High Velocity Kernel File Systems with Bento

Samantha Miller, Kaiyuan Zhang, Mengqi Chen, Ryan Jennings, Ang Chen, Danyang Zhuo, Thomas Anderson; University of Washington, Duke University, Rice University

1.Kernel File System Development is Slow.

- High development and deployment velocity are **critical** to modern cloud systems: on average 650,000 lines of Linux code are added every release cycle for addressing newfound vulnerabilities and for managing new workloads by new devices .
- Linux kernel development is slow: an approach is to directly modify the kernel source code through kernel interface like virtual file system (VFS), while kernel code paths are **complex and easy to accidentally misuse,** meanwhile **debugging** kernel source code is much harder than user level debugging.
- File systems are particularly affected due to advances in storage hardware and new demands by cloud systems.

2.Existing Techniques aren't Sufficient: besides directly modifying the Linux kernel, there are two other approaches to adding functionality to Linux:

- **Upcall (FUSE):** proposed for file systems and I/O devices, to implement new functionality as a user-space server. A stub is left in kernel that converts system calls to upcalls into server; *performance cost for metadata operations, cannot reuse kernel features like disk accesses through the buffer cache.*
- **In-Kernel Interpreter:** uses an interpreter inside the kernel for a dynamically loaded program in a safe language; difficult to implement **larger or more complex pieces** of functionality

High Velocity Kernel File Systems with Bento

Samantha Miller, Kaiyuan Zhang, Mengqi Chen, Ryan Jennings, Ang Chen, Danyang Zhuo
Washington, Duke University, Rice University

1.Kernel File System Development is Slow.

- High development and deployment velocity are **critical** to modern cloud systems: o ... ed every release cycle for addressing newfound vulnerabilities and for managing new v
- Linux kernel development is slow: an approach is to directly modify the kernel sou ... le system (VFS), while kernel code paths are **complex and easy to accidentally misus** ... s much harder than user level debugging.
- File systems are particularly affected due to advances in storage hardware and new demands by cloud systems.

Bug	Number	Effect on Kernel
Use Before Allocate	6	Likely `oops`
Double Free	4	Undefined
`NULL` Dereference	5	`oops`
Use After Free	3	Likely `oops`
Over Allocation	1	Overutilization
Out of Bounds	4	Likely `oops`
Dangling Pointer	1	Likely `oops`
Missing Free	18	Memory Leak
Reference Count Leak	7	Memory Leak
Other Memory	1	Variable
Deadlock	5	Deadlock
Race Condition	5	Variable
Other Concurrency	1	Variable
Unchecked Error Value	5	Variable
Other Type Error	8	Variable

Table 1: Low-level bugs in released versions of OverlayFS, AppArmor, and Open vSwitch Datapath between 2014-2018, categorized as memory bugs, concurrency bugs, or type errors, and the likely effect of each bug on kernel operation.

Such bugs are hard to uncover by testing but can lead to serious impacts on the integrity of the kernel.

2.Existing Techniques aren't Sufficient: besides directly modifying the Linux kernel, there are two other approaches to adding functionality to Linux:

- **Upcall (FUSE):** proposed for file systems and I/O devices, to implement new functionality as a user-space server. A stub is left in kernel that converts system calls to upcalls into server; *performance cost for metadata operations, cannot reuse kernel features like disk accesses through the buffer cache.*
- **In-Kernel Interpreter:** uses an interpreter inside the kernel for a dynamically loaded program in a safe language; difficult to implement **larger or more complex pieces** of functionality

High Velocity Kernel File Systems with Bento

Samantha Miller, Kaiyuan Zhang, Mengqi Chen, Ryan Jennings, Ang Chen, Danyang Zhuo, Thomas Anderson; University of Washington, Duke University, Rice University

- **Bento,** a framework for high velocity development of Linux kernel file systems.
- **Safe Interfaces for a Safe Language**
  - Safe interfaces provided by Bento enable the file system to be written in safe Rust.
  - BentoFS and libBentofs translate the VFS interface to a safe Rust interface inspired by message passing interfaces This interface is based on the FUSE low-level interface.
  - LibBentoKS provides safe wrappers around kernel services such as the buffer cache, kernel lock implementations, and the kernel TCP stack.
- **Live Upgrade**
  - A **live upgrade component** in BentoFS manages the **live upgrade process**, allowing the old file system to transfer state to the new one and swapping function pointers.
- **Userspace Execution**
  - Bento file system can be run in user-space without any changes to the code.
  - Most interfaces provided by libBentoFS and libBentoKS are identical to existing user-space interfaces.
  - File system can be compiled to run in user-space using a build flag.

FAST’21 → Advanced File Systems

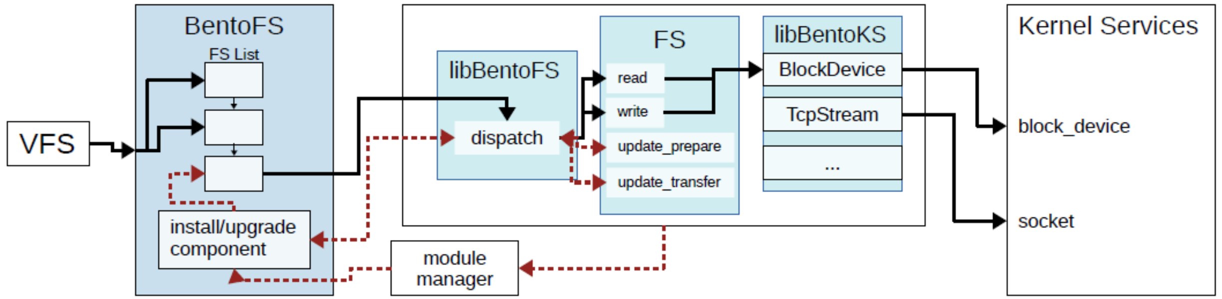

Figure 1: Design of Bento. Shaded components are parts of Bento. BentoFS is in C. The other shaded components are in Rust. Solid black lines represent the common-case operation pathway, detailed in §3.2 and §3.3. Dashed red lines represent the install/upgrade pathway and are described in §3.4

High Velocity Kernel File Systems wit

Samantha Miller, Kaiyuan Zhang, Mengqi Ch
Washington, Duke University, Rice University

- **Bento,** a framework for high velocity d
- **Safe Interfaces for a Safe Language**
  - Safe interfaces provided by Bento enable the file system to be written in safe Rust.
  - BentoFS and libBentofs translate the VFS interface to a safe Rust interface inspired by message passing interfaces This interface is based on the FUSE low-level interface.
  - LibBentoKS provides safe wrappers around kernel services such as the buffer cache, kernel lock implementations, and the kernel TCP stack.
- **Live Upgrade**
  - A **live upgrade component** in BentoFS manages the **live upgrade process**, allowing the old file system to transfer state to the new one and swapping function pointers.
- **Userspace Execution**
  - Bento file system can be run in user-space without any changes to the code.
  - Most interfaces provided by libBentoFS and libBentoKS are identical to existing user-space interfaces.
  - File system can be compiled to run in user-space using a build flag.

Scalable Persistent Memory File System with Kernel-Userspace Collaboration

Youmin Chen, Youyou Lu, Bohong Zhu, Andrea C, Arpaci-Dusseau, Remzi H, Arpaci-Dusseau, Jiwu Shu; Tsinghua University, University of Wisconsin – Madison.

Byte-addressable PM emerges while current file system cannot efficiently handle it, especially for **scalability**.

- ***Kernel-level file systems,*** are part of the OS and applications need to trap into the kernel to access, where syscalls and the virtual file system (VFS) still incur non-negligible software overhead.
- ***Userspace file systems,*** are deployed in userspace to directly access file data with better performance, while the scalability is ignored (not handled well). *Since centralized components still exist, it still rely on the kernel to enforce coarse-grained allocation and protection.*
- ***hybrid file systems,*** behave better for performance, while the scalability is bad.

Criterions for PM-aware file system:

- ***Multicore scalability.*** The VFS that are adopted by Kernel-level fs is the global bottleneck: the throughput is almost unchanged as increasing threads since VFS needs to acquire the lock of the parent directory; The centralized components or logging methods adopted by uer-space fs behaves as the bottleneck under concurrency;
- ***Software overhead.*** Mainly caused by VFS or syscall of Linux kernel operation, which are invoked by both kernel/user fs. (*Current hybrid fs still requires kernel fs to handle metadata, which behaves just like the kernel fs.)*
- ***Other issues.*** Misused pointers, data visibility delaying, requiring data protection hardware.

In a word, it is hard to achieve high scalability and low software overhead with existing file system designs.

Scalable Persistent Memory File System with Kernel-Userspace Collaborat

Youmin Chen, Youyou Lu, Bohong Zhu, Andrea C, Arpaci-Dusseau, Remzi H, Arpaci-Dus
University of Wisconsin – Madison.

	Category	NOVA	Aerie/Strata	ZoFS	SplitFS	KucoFS
	Category	Kernel	Userspace		Hybrid	
❶ Scalability	Metadata	Medium (§5.2.1)	Low (§5.2.1)	Medium (Fig. 7g in [13])	Low (§5.2.1)	High (§5.2.1)
❶ Scalability	Read	Medium (§5.2.2)	Low (§5.2.2)	High	Low (journaling in Ext4)	High (§5.2.2)
❶ Scalability	Write	Medium (§5.2.3)	Low (§5.2.3)	Medium (Fig.7f in [13])		High (§5.2.3)
❷	Softeware overhead	High	Low	Medium (`sigsetjump`)	Medium (metadata)	Low
❸ Other issues	Avoid stray writes	✓	✗	✓	✗	✓
❸ Other issues	Read protection	POSIX	Partition	Coffer	POSIX	Partition
❸ Other issues	Visibility of updates	Immed-iately	After batch/ After digest	Immed-iately	append: After sync	Immed-iately
❸ Other issues	Hardware required	None	None	MPK	None	None

Table 1: Comparison of different NVM-aware file systems.

Byte-addressable PM emerges while current file system cannot efficiently handle

- ***Kernel-level file systems,*** are part of the OS and applications need to trap into the kernel system (VFS) still incur non-negligible software overhead.
- ***Userspace file systems,*** are deployed in userspace to directly access file data with better not handled well). *Since centralized components still exist, it still rely on the kernel to enforce coarse-grained allocation and protection.*
- ***hybrid file systems,*** behave better for performance, while the scalability is bad.

Criterions for PM-aware file system:

- ***Multicore scalability.*** The VFS that are adopted by Kernel-level fs is the global bottleneck: the throughput is almost unchanged as increasing threads since VFS needs to acquire the lock of the parent directory; The **centralized components** or logging methods adopted by uer-space fs behaves as the bottleneck under concurrency;
- ***Software overhead.*** Mainly caused by VFS or syscall of Linux kernel operation, which are invoked by both kernel/user fs. (*Current hybrid fs still requires kernel fs to handle metadata, which behaves just like the kernel fs.)*
- ***Other issues.*** Misused pointers, data visibility delaying, requiring data protection hardware.

In a word, it is hard to achieve high scalability and low software overhead with existing file system designs.

Scalable Persistent Memory File System with Kernel-Userspace Collaboration

Youmin Chen, Youyou Lu, Bohong Zhu, Andrea C, Arpaci-Dusseau, Remzi H, Arpaci-Dusseau, Jiwu Shu; Tsinghua University, University of Wisconsin – Madison.

Kuco, a **k**ernel-**u**serspace **co**llaboration architecture, to achieve both **direct access performance** and **high scalability**.

- ***Basic Principle: combine good properties of both kernel and userspace file systems, while delivering high scalability.***

  It follows a classic **client/server model** with two components: a userspace library (**Ulib**) to provide basic file system interfaces, and a trusted thread (**Kfs**) placed in the kernel to process requests sent by Ulib and perform critical updates (e.g., metadata).
- ***Main idea: fine-grained task division, offloading time-consuming tasks from Kfs (kernel/server) to Ulib (userspace/client), avoiding kernel bottlenecks, for which:***
  - ***Collaborative Indexing*** **for metadata scalability**, it allows Ulib to perform pathname resolution before sending requests to Kfs, so that Kfs can update metadata items directly with the pre-located addresses provided by Ulib. *(introduce EBR for read consistency.)*
  - ***Two-level Locking*** **for data scalability (write),** it coordinates concurrent writes to shared files, where Kfs manages a write lease for each file and assigns it to the process that intends to open the file, and threads within this process lock the file with a range-lock completely in userspace.
  - ***Versioned Reads*** **for direct reads,** it achieves direct reads even without interacting with Kfs, despite the presence of concurrent writers.
- ***Also enforcing data protection and improve baseline performance***. Kuco maps the PM space into user-space in read-only mode to prevent buggy programs from corrupting file data.

Scalable Persistent Memory File System with Kernel-Userspace Coll

Youmin Chen, Youyou Lu, Bohong Zhu, Andrea C, Arpaci-Dusseau, Remzi H, Arpa
University of Wisconsin – Madison.

Kuco, a **k**ernel-**u**serspace **co**llaboration architecture, to achieve both **direct**

- ***Basic Principle: combine good properties of both kernel and userspace file syste***

  It follows a classic **client/server model** with two components: a userspace library (**Ulib**) to

  placed in the kernel to process requests sent by Ulib and perform critical updates (e.g., meta

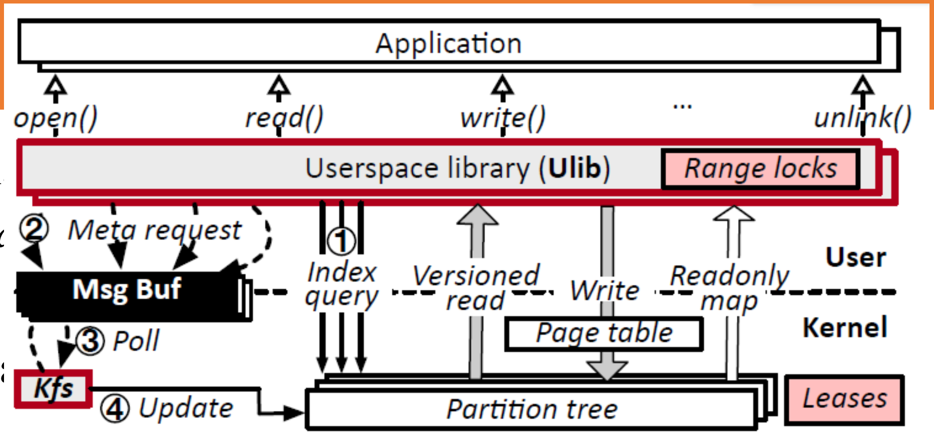

Figure 2: The Kuco architecture. metadata updates (①-④): Ulib interacts with Kfs via *collaborative indexing*; read: direct access via) *versioned read*; write: direct access based on a *three-phase write protocol* and *two-level locking* for concurrency control.

- ***Main idea: fine-grained task division, offloading time-consuming tasks from Kfs (kernel/server) to Ulib (userspace/client), avoiding kernel bottlenecks, for which:***
  - ***Collaborative Indexing* for metadata scalability**, it allows Ulib to perform pathname resolution before sending requests to Kfs, so that Kfs can update metadata items directly with the pre-located addresses provided by Ulib. *(introduce EBR for read consistency.)*
  - ***Two-level Locking* for data scalability (write),** it coordinates concurrent writes to shared files, where Kfs manages a write lease for each file and assigns it to the process that intends to open the file, and threads within this process lock the file with a range-lock completely in userspace.
  - ***Versioned Reads* for direct reads,** it achieves direct reads even without interacting with Kfs, despite the presence of concurrent writers.
- ***Also enforcing data protection and improve baseline performance***. Kuco maps the PM space into user-space in read-only mode to prevent buggy programs from corrupting file data.

FAST’21 → Advanced File Systems

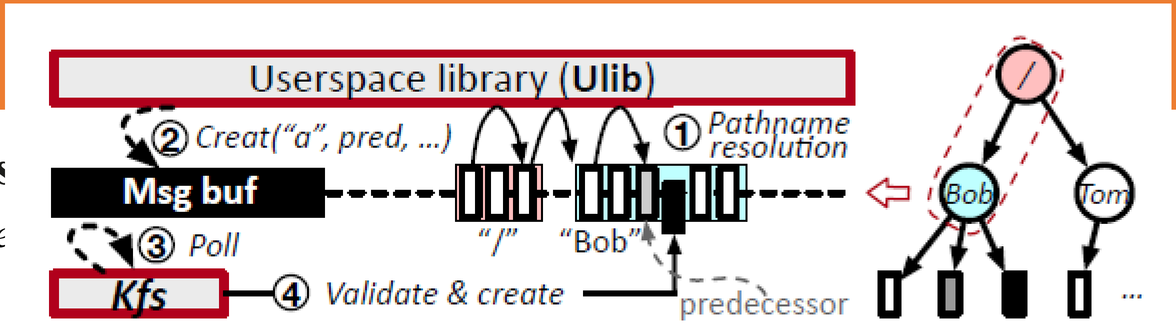

Figure 3: Creating a file (①-④) with *collaborative indexing*.

Scalable Persistent Memory File System with Kernel-Users

Youmin Chen, Youyou Lu, Bohong Zhu, Andrea C, Arpaci-Dusseau, Re
University of Wisconsin – Madison.

Kuco, a **k**ernel-**u**serspace **co**llaboration architecture, to achieve bo

- ***Basic Principle: combine good properties of both kernel and userspace file syste***

  It follows a classic **client/server model** with two components: a userspace library (**Ulib**) to

  placed in the kernel to process requests sent by Ulib and perform critical updates (e.g., meta

Figure 2: The Kuco architecture. metadata updates (①-④): Ulib interacts with Kfs via *collaborative indexing*; read: direct access via *versioned read*; write: direct access based on a *three-phase write protocol* and *two-level locking* for concurrency control.

- ***Main idea: fine-grained task division, offloading time-consuming tasks from Kfs (kernel/server) to Ulib (userspace/client), avoiding kernel bottlenecks, for which:***
  - ***Collaborative Indexing* for metadata scalability**, it allows Ulib to perform pathname resolution before sending requests to Kfs, so that Kfs can update metadata items directly with the pre-located addresses provided by Ulib. *(introduce EBR for read consistency.)*
  - ***Two-level Locking* for data scalability (write),** it coordinates concurrent writes to shared files, where Kfs manages a write lease for each file and assigns it to the process that intends to open the file, and threads within this process lock the file with a range-lock completely in userspace.
  - ***Versioned Reads* for direct reads,** it achieves direct reads even without interacting with Kfs, despite the presence of concurrent writers.
- ***Also enforcing data protection and improve baseline performance***. Kuco maps the PM space into user-space in read-only mode to prevent buggy programs from corrupting file data.