Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Master's Thesis Nr. 43

Systems Group, Department of Computer Science, ETH Zurich

Towards a File-System Service for the Barrelfish OS

by

Manuel Stocker

Supervised by

Prof. Timothy Roscoe, Dr. Kornilios Kourtis

January 2012–July 2012

inf | Informatik
Computer Science

With the advent of non-volatile memory technologies such as memristors and phase change memory, the performance gap between main memory and persistent storage is closing. Where file systems have traditionally made optimizations towards slow, block oriented access, new approaches will have to be taken in order to exploit the low-latency, byte-addressable interfaces of non-volatile ram.

This thesis introduces Goby, a layered approach to storage systems geared towards highly parallel use. It is based on versioned B+Tree structures, supporting transactional semantics and snapshot isolation. It focuses on main memory performance characteristics and assumes that persistent storage is implemented in the memory path.

Goby is built to support the explicit message passing architecture of the Barrelfish research operating system, but uses shared memory for the implementation presented in this thesis. This imposes a restriction in only supporting cache-coherent architectures which can be lifted by modifications to Goby's lowest layer.

The thesis evaluates the performance characteristics of Goby compared to a recent version of `ramfs` used in Linux for a chosen set of workloads. This comparison is done with semantics similar to POSIX due to `ramfs` being a POSIX-compliant file system.

Contents

1 Introduction

1.1 Motivation

File systems have traditionally been geared towards hard disks using magnetic platters for data storage. Their characteristics such as block oriented access, large seek times and the magnitudes of difference of read/write throughput compared to main memory have been defining for the file-system's architecture. With recent advances in storage technologies such as solid-state disks and non-volatile RAM, the performance gap between memory and non-volatile storage is narrowing. The design of future file systems can therefore exploit optimisations that current in-memory storage systems use but are not possible on current hard disks.

Additionally, the API of file systems has historically been little more than an abstraction of byte-level access to files organized in directories. Traditionally, there is no notion of transactional contexts. Programmers wishing to perform atomic operations in current systems have to use tricks such as using the *rename* operation.

Finally, technological advances in performance have shifted towards more parallel processing elements instead of increasing performance of a single processor. This leads to higher concurrency in all aspects of a modern OS. To fully leverage the parallel potential, a file-system will have to provide the programmers with a way of specifying parallel contexts and intelligently distribute the workload among processing cores.

1.2 Non-Volatile Memory

Disk-based storage devices are already outperformed by newer technologies such as solid-state disks by orders of magnitude. Solutions based on Flash memory are used to speed up applications depending on high I/O throughput to persistent storage [1]. Those Flash-based solutions still suffer from the memory gap and only provide block-level access. However, there are multiple technologies looking promising to close this gap, such as memristors [8] and phase change memory (PCM) [12]. Storage based on such non-volatile memory technologies could be integrated directly into the memory path analogous to DRAM with similar access characteristics [6] but allowing persistent storage of data.

These types of memory often have a relatively limited number of write-cycles and are thus better suited for long-term storage instead of replacing DRAM completely. Venkataraman et. al. [15] have already studied a versioned tree structure focused on consistent and durable storage in a byte-addressable NVRAM.

1.3 Barrelfish

Barrelfish [3] is a research operating system created at ETH Zurich in collaboration with Microsoft Research. It introduces the multikernel, a kernel architecture focused on explicit message passing instead of using sharing for kernel data structures. State is replicated instead of implicitly shared. This approach can scale better considering hardware is looking more like a distributed system with the trend towards more processing cores and more sophisticated interconnects.

The existing file system implementation in Barrelfish is an in-memory file system built in a centralized manner: The `ramfsd` service allocates memory for file storage and directory structure. Access is provided via an RPC interface based on message passing. The bulk-transfer library is used to transfer data payloads to and from the service. RPC requests are handled through a queue, serializing individual operations. Modifications are done locally in a single thread handling incoming messages. The current implementation is obviously not very sophisticated as it was built to address the basic need for a file system.

1.4 Goals

This thesis explores the aspects and building blocks of a modern file system targeted at non-volatile memory. It's focus is on providing data structures and algorithms for concurrent access in a low-latency environment. As a consequence, it will avoid trade-offs traditionally taken because of block-oriented, high-latency access to legacy disk drives.

Since no commercial non-volatile RAM systems integrated on the memory path are currently available, the system developed in this thesis will use main memory as a model of the access characteristics of such storage-class memory devices [6].

The design of the architecture will also keep the explicit message passing approach of Barrelfish in mind. The parts that require sharing will be structured in a way that allows straight-forward replacement with components using message driven synchronization and replication. As such, architectures without shared memory can be supported by a suitable implementation.

2 Problem Analysis

2.1 Requirements

A file system has to map objects of various types to a binary representation suitable to be stored in an area of continuous storage. This area might have restrictions on access patterns such as the block size of hard disks. Examples for objects in a file system are directories and files. Files map a range of positions to a range of values. Directories map filenames to files and sub-directories. Usually it is desired to have meta-data such as access permissions, modification times etc. for both files and directories.

Early file systems (such as FAT and ext2) map directories to files by simply storing a list of pairs consisting of the filename and a pointer to the data in a file with the name of the directory. This approach requires a linear scan over the directory contents to find a given filename and as such scales linearly with the number of files in the directory. To improve performance, different approaches have been taken for large directories, such as HTree [11] used in ext3 and ext4. Similarly, file systems have transitioned from using linked lists to chain individual data blocks to more sophisticated data structures such as B-trees.

To summarize, a modern file system should provide fast...

- lookup of files by name.
- write and read access to arbitrary ranges in files.
- file/directory creation/removal.

These requirements should not degrade under concurrency to allow scaling to large numbers of concurrent operations. Since the goal of this thesis is to develop an in-memory file system, an efficient design decision can directly be taken instead of using a disk-friendly design and building caches in memory to gain the necessary performance.

2.2 Change Visibility

The semantics of a file system with respect to change-visibility can influence the design of data structures greatly, since different guarantees have to be provided. The POSIX [2] standard mandates that all changed bytes are visible for any successful subsequent reads until overwritten by another write call. This implies atomic writes either visible completely or not at all. NFS is on the other side of the spectrum, allowing for non-coherent caching among clients [5]. A write operation could therefore be only locally visible for an extended duration. A possible middle ground is to require changes to be visible upon closing the file or after explicit flushing.

2.3 Approaches to Concurrency

Traditionally mutexes have been used to ensure serialized modification of shared data structures. Linux used to have a single mutex in the kernel which has been abandoned in favour of fine-grained mutexes, spinlocks and read-copy-update (RCU) where possible. However, RCU implementations depend on determining the grace-period in which all participating CPUs have performed a context-switch and therefore will not hold any references to stale data anymore [10]. This leads to increased latency proportional to the number of concurrent readers [10, p. 5]. The gained benefit is however that concurrent readers always see a consistent state without having to acquire locks. RCU can use an inter-processor interrupt to force a context-switch in foreign processors to reduce latency.

Versioned data structures extend this principle by allowing participants to create their own version of the data structure. If we restrict the number of concurrent writers to one, this directly translates to RCU. When allowing multiple concurrent writers however, we need to merge their individual versions to form a new version that further versions can be based on and readers can use. Meld [13] is a system that makes extensive use of this principle. The write operations are stored in a shared intent log. Every process in the system is able to determine which transactions can be committed based on this log and will apply them do derive a common state. Since all processes share the same log, every process will make the same decisions without the need for explicit synchronisation apart from serialized appending to the shared log. The authors of Meld argue that the node handling the log can be scaled up well enough.

2.3.1 Multi-versioned B+Trees

Multi-versioned B+Trees use a shadowing scheme to implement generational access to a tree. Examples of their use are the WAFL [7], ZFS [4] and BTRFS file systems, among others.

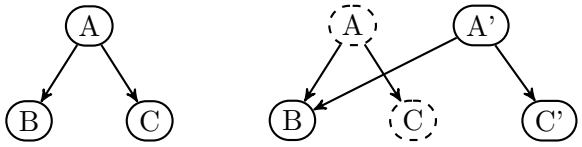


Figure 2.1: B+Tree shadowing

An example of shadowing is shown in figure 2.1. The tree initially contains the root node A and the leaves B and C. A write to a value stored in leaf C results in a copy of leaf C with the modification in leaf C'. Furthermore, since the link to leaf C' has to be inserted in the node A, it is also shadowed to produce the new root node A'.

Rodeh presents algorithms for a shadowing B-Tree to implement writable snapshots for a file system [14]. The same approach can also be used to handle concurrency: For every concurrent process, one can create a clone and use it to do modifications. If there are no concurrent writers, committing these modifications is as simple as updating the

root pointer to the new tree. The main challenge with concurrent writers is to merge these changes back to the shared state respecting concurrent writes from other processes.

Another advantage of versioned trees is the life-cycle of data. Elements of the tree are only written to once and never modified in place. Once elements are not referenced anymore, they can be reclaimed and the corresponding memory can be reused to store new elements. This is an important aspect when having to explicitly share memory for example on non-cache-coherent architectures. If a process does not already have a copy of an element, one can be created and cached until the element is reused.

2.4 Linux in-memory File System: `ramfs`

Linux uses `ramfs` as the primary in-memory file system. It could be easily extended to use NVRAM as backing storage to make data persistent. It is built as a thin layer over the directory entry (dentry) and page caches, not allowing the corresponding entries to be evicted from the respective caches.

The dentry-cache is a hash table containing `dentry` structures. These represent directory entries and form a tree. As such, every `dentry` contains a pointer to its parent as well as a hashed list of children.

The page-cache keeps pages of an address space in a radix-tree. Updates are performed with RCU. The data of each file is handled as an address space associated with an inode representing the file and storing its meta-data.

3 Approach

3.1 Assumptions

This thesis assumes that file systems are not shared among architectures with different endianness or register size to avoid having to write a lot of glue- and conversion code. Handling such scenarios would be possible by defining an on-disk endianness and datatype sizes, as well as implementing wrappers that perform conversion when necessary.

The file system will be constructed entirely in main memory. This implies that the storage device is either integrated into the memory path or it is acceptable to have an intermediary layer which will retrieve and store accessed memory regions on demand. This can be realized by a separate process handling transfers between main memory and the underlying storage device.

For simplicity, the algorithms in this thesis will assume the availability of an atomic compare-and-swap or an equivalent instruction as well as cache-coherency. However to avoid being dependant on this particular type of architecture, abstractions will be made to provide points of integration for sharing mechanisms beyond cache-coherency.

3.2 Versioning

Since trees can exist in multiple versions, the different incarnations form a directed graph $VG = (V, E)$. Creating a new version $v' \in V$ based on version $v \in V$ adds an edge $(v, v') \in E$. Merging two versions $v_x \in V$ and $v_y \in V$ to produce version v_{xy} will in turn add two edges (v_x, v_{xy}) and (v_y, v_{xy}). Let $x < y$ denote the transitive relation such that $x < y \rightarrow (x, y) \in E$, ie. y is a version directly or indirectly based on x.

Goby will use positive integers to represent versions. This loses the complete graph while still maintaining the relation $<$ which directly translates to the *smaller than* relation on integers. Multiple distinct versions can share the same version number. In order to avoid ambiguous versioning, Goby keeps a *blessed* version that is shared among processes. New versions may only be based on a blessed version and merges may only be performed with blessed versions. A version may only be blessed if it is directly based on the most recent blessed version. Therefore for a blessed version v_y the following holds: $B(v_y) \rightarrow \exists v_x(B(v_x) \wedge (v_x, v_y) \in E)$. This implies that for every two distinct versions v_x and v_y, a blessed version smaller than both is a common ancestor.

Figure 3.1 illustrates an example with two concurrent processes. Both processes create their own versions based on the shared blessed version. Process P_2 is first in promoting its version to be the new blessed version. Since its version satisfies the criteria, it can be

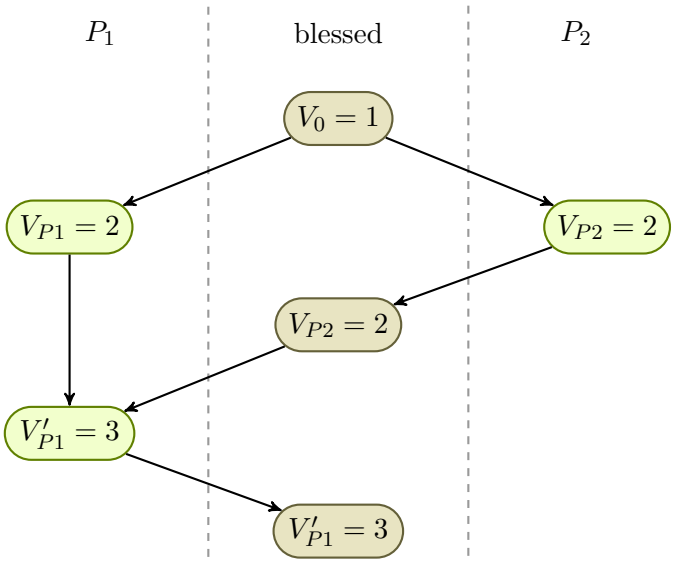


Figure 3.1: Example sequence of versioning with two processes.

blessed without merging. Process P_1 on the other hand needs to satisfy the requirement of having the most recent blessed version as a direct ancestor and therefore has to merge with the new blessed version V_{P2} to produce a candidate for the final blessed version. These requirements directly translate to the flat version numbers: Since $V_{P2} = 2$ is not smaller than $V_{P1} = 2$, a merge has to be performed. The resulting $V'_{P1} = 3$ satisfies the condition with $V_{P2} < V'_{P1} \Leftrightarrow 2 < 3$ and therefore can be blessed.

3.3 Architecture

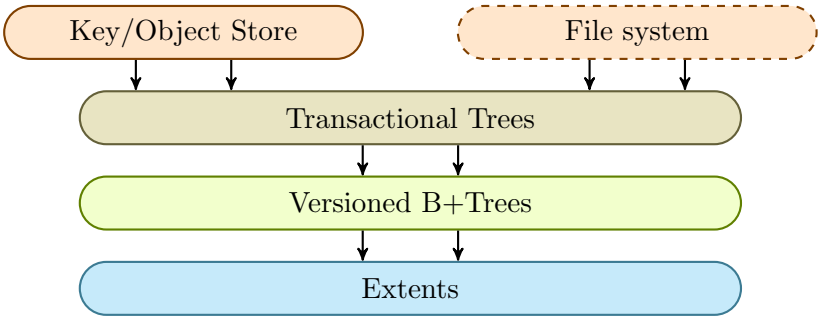


Figure 3.2: Architectural Overview

Goby is structured in layers. Figure 3.2 shows the individual layers. Higher layers of the system only depend on lower layers and their exposed API. This allows different implementations of a lower layer without having to change upper layers.

3.3.1 Extents

The extent layer is responsible for providing a number of continuous memory regions referred to as extents. Each one is identified by a unique extent ID. The API offers allocation and access to meta-data necessary for the upper layers. These are:

- **Version**: Each extent has a version number according to the versioning scheme outlined in section 3.2.
- **Reference count**: In order to be able to perform garbage collection, every extent has a reference number denoting the number of references to this extent. Both references from structures in other extents as well as references from processes currently reading an extent count towards the reference count. It is therefore crucial that reference count modifications are performed atomically.
- **Type**: The type of an extent hints at the data structure stored in the extent. It is only set upon allocation of the extent and never changed.

The API of the extent functionality is as follows:

- `alloc_any(out extent_id, type, version)` Allocates an extent and returns it's ID in the `extent_id` parameter. The extent layer is responsible for finding a free extent, allocating it and setting its type and version to the supplied values.
- `inc_refcount(extent_id)` Increment the reference count of the extent with the given ID. This function **must** return an error if the extent is concurrently being deallocated or has a reference count of zero. It does however not have to guarantee that an extent has not been garbage collected and reallocated.
- `dec_refcount(extent_id)` Decrement the reference count of the extent with the given ID. Whenever the reference count reaches zero, the extent **must** be deallocated at some point in the future. Decreasing the reference count below zero should result in an error.
- `{get|set}_{version|type}(extent_id, in|out value)` These functions should retrieve resp. set the desired property. Behaviour is undefined if the given extent is not allocated.
- `to_ptr(extent_id, out ptr)` Get a pointer to the contents of the given extent. This function should be replaced with a pair of read and write functions in further work based on this thesis in order to differentiate between reading and writing.

The internal layout of data and the storage method for extents is left to the implementation, although extents have to be present in main memory after a call to `to_ptr` for the current API. This is a shortcut taken to avoid having no-op read/write wrappers.

3.3.2 Versioned B+Tree

This layer implements a versioned B+Tree stored in the extents provided by the extent layer. The API consists of operations for creating trees as well as inserting, finding and deleting entries:

- `create(out extent_id, version)` Create a new tree with the given initial version. The implementation is responsible to allocate an extent from the extent layer. The returned extent shall contain an empty tree.

- `find(extent_id, key, out value)` Find the value associated with the given key. If the tree does not contain the given tree return a not found error.

- `insert(inout extent_id, key, value, version)` Insert a key/value pair into the tree rooted in the given extent. The tree shall be shadowed on demand to produce the given version. If shadowing is necessary and thus results in a change of the root for the tree, the extent ID of the new root will be returned in `extent_id`. If the insertion will replace an existing key, the implementation has to ensure that the reference count is adjusted properly.

- `delete(inout extent_id, key, version)` Remove the given key from the tree. The tree shall be shadowed on demand similar to insertions. If the tree does not contain the key, return a not found error.

- `traversal_init(extent_id, start_key, end_key, out handle)`
  Initialise a traversal of the tree rooted in the given extent from (including) the start key to (including) the end key. If the tree does not contain any key in the requested range, return a not-found error.

- `traversal_next(handle, out key, out value, out finished)`
  Return the next key/value pair in sorted order. The implementation shall set `finished` to `true` and free the handle if there are no more keys in the specified range.

- `garbage_handler(extent_id)` The garbage handler to be called if a tree node is deallocated. The implementation shall decrease the reference count on any referenced extents.

- `merge(ancestor, bottom, top, out merged)` Merge the changes in the tree given in `top` relative to the common ancestor given in `ancestor` into the tree given in `bottom`. The root extent of the resulting tree is given in `merged`.

Since the tree is only used to store references to extents, the values have to be IDs of allocated extents. The implementation needs to ensure correct reference counts during operations such as insertion and shadowing.

3.3.3 Transactions

The transaction layer is responsible for handling the concurrency aspects of Goby. A blessed version is kept and shared between participating processes. Transactions are realized as an extension of the versioned-tree functionality. When beginning a new transaction. A new version of the tree is created based on the currently blessed version. Operations inside the context of the transaction are applied to this version of the tree. Committing will merge changes with a descendant of the base tree if necessary (i.e. another process has committed a new version of the tree) and shares a new blessed version iff there are no conflicts.

The API is an extension of the tree's API:

- `begin(extent_id, out handle)` Start a transaction based on the tree rooted in the given extent and return a handle for the created transaction. The user has to hold a reference to the given extent in order to avoid concurrent deallocation during creation of the transaction. The implementation shall acquire its own references if needed.

- `free(handle)` Free a transaction and release all resources and references held by it. Corresponds to an abortion of the transaction.

- `commit(handle, inout extent_id)` Commit the transaction. The given extent can contain the tree the transaction was based on or a descendant thereof. If a descendant is given, the implementation has to ensure the absence of conflicts and merge the changes with the changes in the descendant. The extent holding the new root shall be returned.

- `nested_transaction(handle, key, inout nested_handle)`
  Create a nested transaction for the tree stored at the extent pointed to by the entry in the given key. Committing or aborting a transaction shall also commit respectively abort all nested transactions.

The API also contains the same functions as the tree's API where the `extent_id` parameters have been replaced with the transaction handle. For example, in case of the `insert` function, the transactional version translates to: `insert(handle, key, value)`.

3.3.4 Key/Object Store

Based on transactional trees, we can build a shared, transactional key/object store with a simple API:

- `init()` Initialize the key/object store. This method should be called exactly once.

- `begin(out handle)` Create a new transaction and return its handle.

- `commit(handle)` Commit the transaction and free its handle.

- `read(handle, key, data, position, length)` Read from the object at the given key into the supplied data pointer, starting at the given position and for the given length of bytes. The read shall be performed in the context of the transaction given by the handle. The object is implicitly considered to be initialized to zero for the maximum possible size. This implies that reading from a location that has never been written to will be considered consisting of zeroes.
- `write(handle, key, data, position, length)` Write to the object at the given key. Data is taken from the supplied data pointer. Writing starts at the given position in the object and shall be done for the given length of bytes.
- `create(handle, key)` Create an empty object at the given key.

3.3.5 Filesystem

Goby does not yet include a complete file system layer. The basic operations used in a file system are however present in the key/object store. Files loosely translate to objects and the key-object mapping can be considered a basic directory structure. Still, section 4.5 presents ideas on how a complete file system could by implemented using the primitives Goby already includes.

The API for a hypothetical file system on Goby can follow the POSIX standard by carefully choosing the scope of transactions. However, an interface exposing Goby's transactional facilities might be more useful for consumers, alleviating the need for tricks such as using the atomic `rename` to emulate transactional behaviour.

3.4 Limitations

Goby assumes that all extents are of the same size to avoid having to complicate code with handling of special cases. For example, this limitation makes splitting and joining B+Tree nodes simpler, since all nodes are of the same size. It could be removed by handling these special cases and providing infrastructure for specifying a desired size on extent allocation or alternatively implement splitting and joining algorithms that can handle $m \to n$ type operations instead of $1 \to 2$ and $2 \to 1$. It is doubtful whether the increase in code complexity in both the allocator and the edge-case handling of the individual data structures would be worth the flexible size.

Goby also limits reads and writes in its key/object store implementation to multiples of the extent size. This avoids having to merge writes inside a single extent. To remove this limitation, logging and merging strategies would have to be adjusted to properly merge concurrent writes to subsets of an extent.

Note that the key/object store does not include methods for deletion or appending data. The absence thereof is because of time constraints and not an inherent limitation of the approach presented in this thesis.

4 Implementation

This chapter details some aspects of Goby's implementation. This section will first look at the extent layer and continue on the way up the layers and highlight algorithms and considerations. Goby is implemented in C and uses SConstruct to build. The code is written for x86-64 and needs adjustments to run on different architectures. To avoid clutter in code extracts and examples, the `pfs_` (*parallel file system*) prefixes in function and type names have been removed.

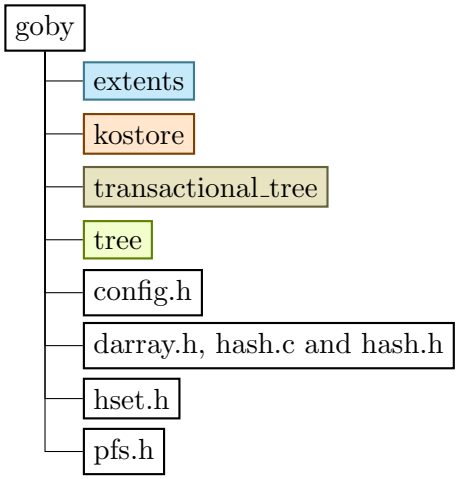


Figure 4.1: Overview directory structure

Figure 4.1 shows an overview of Goby's directory structure. Contents are:

- `extents`, `kostore`, `transactional_tree` and `tree` contain the implementation of the respective layers of Goby.
- `config.h` Contains C preprocessor definitions for different configurable values. Defaults can be overridden by either defining them on the compiler's command line or in the file `config_custom.h` which will be automatically created when building.
- `darray.h`, `hash.h` and `hash.c` are implementations of a dynamic array and hashing algorithms taken from the CCAN.
- `hset.h` Contains an implementation of a hashset that stores values instead of pointers.
- `pfs.h` Contains various frequently used macros and types, as well as error codes.

4.1 Extent Layer

Figure 4.2: Internal memory organisation

Goby's extent layer simply has to be supplied with a chunk of main memory where it stores all its data structures. Figure 4.2 shows the internal organisation: Space for the necessary amount of meta-data entries is reserved at the beginning, dividing the rest into equally sized extents of a predefined size. Every extent has a corresponding entry in the meta-data. As such, the number of meta-data entries is defined by the maximum number of extents: $N_{max} = \lfloor \frac{area\_size}{extent\_size} \rfloor$. The number of reserved extents is $R = \lceil \frac{N_{max}}{size(metadata\_entry)} \rceil$. The resulting number of usable extents is $N = N_{max} - R$. The structure used for meta-data entries is:

```
struct extent_header {
    volatile uint64_t allocated;
    volatile extent_version_t version;
    volatile int64_t refcount;
    volatile enum extent_type_t type;
};
```

During initialization, all meta-data entries are set to zero, conveniently setting all extents to unallocated and their type to `free`.

4.1.1 Extent Allocation

The extent allocator keeps an internal state consisting of the position next to the last allocation. When allocating a new extent, it will start searching at the saved position until a free extent is found. Searching continues at the first extent upon going past the last one. Allocation will fail if every position has been unsuccessfully checked.

When a free extent is encountered, the extent allocator will try setting `allocated = 1` with an atomic CPU instruction. If another process was faster in acquiring the extent, the search is continued.

Extra care has to be taken in concurrent situations. If multiple allocator calls take place in parallel, performance can suffer heavily if they are all trying to allocate the same set of extents. Therefore, the function `init_extent_allocator` can be used to set starting positions dividing the extent space into equal chunks proportional to the number of concurrent processes.

There still exists the risk that the allocator positions will converge eventually due to the slower allocation if they allocate close to each other. The allocator has an additional *private allocation* mode which restricts each concurrent process to a disjoint subset of extents which will prevent contention between concurrent allocators, at the expense of a reduced set of available extents for each process.

4.1.2 Reference Counting

Modifications of the reference count have to be applied atomically to avoid race-conditions when reaching zero and subsequently having to garbage collect the extent. We cannot use `fetch_and_add` for this reason, since there is a race-condition if one process increases the reference count while another process decreases it to zero and subsequently garbage collects the extent. This sequence of events leaves a garbage-collected extent with a reference count greater than zero, violating the requirement that extents with positive reference counts cannot be de-allocated.

However, we can use `compare_and_swap` inside of a loop to achieve our goal. Algorithm 4.1 shows incrementing and algorithm 4.2 decrementing of the reference counter. While incrementing is straight-forward, decrementing has an additional state where the reference count is set to -1 while running the garbage handler for the extent type in order to de-allocate it.

Algorithm 4.1 Increment reference count

```
while true do
  if refcount ≤ 0 then
    return false
  end if
  if compare_and_swap(&refcount, refcount, refcount + 1) = refcount then
    return true
  end if
end while
```

Algorithm 4.2 Decrement reference count

```
while true do
  if refcount ≤ 0 then
    return false
  end if
  if refcount = 1 then
    if compare_and_swap(&refcount, refcount, -1) = refcount then
      collect_garbage()
      refcount ← 0
      return true
    end if
  else
    if compare_and_swap(&refcount, refcount, refcount - 1) = refcount then
      return true
    end if
  end if
end while
```

Garbage handlers can be registered by calling `register_garbage_handler(type, handler)`

which will simple store the given function pointer in the array of garbage handlers at the type's index. These handlers are necessary to adjust reference counts on referenced extents. A tree node for example will need to decrement the reference count on any child nodes.

The allocator's initial position can be randomized by calling `init_extent_allocator` with a number that will be passed to `srand`. A random starting position is then chosen: `rand() % num_extents()`. In *private allocation* mode, the position is not randomized but set to be the i^{th} bucket of $\frac{N}{i}$ extents. The range of the allocator is restricted to this bucket.

4.1.3 Extent Stack

The extent layer also contains a straight-forward implementation of a stack of extent IDs with the standard methods `create`, `push`, `pop`, `peek` and `free`. It is used whenever a path from the root to a leaf has to be stored (eg. in traversals).

4.1.4 Problems Encountered

The first prototype of Goby divided the entire memory area into extents of a fixed size, storing meta-data such as the reference count, version and type in the first few bytes of every extent. There are severe disadvantages considering caching, since these memory addresses will map to the same set of cache lines and thus inducing issues with false sharing. The prototype also used a queue for the free extents instead of a simple bitmap, making multiple freeing of an extent hard to debug.

4.2 Versioned B+Tree

Goby contains an implementation of a versioned B+Tree that can be compiled for keys and values of arbitrary size. However, since the tree is always used to map to extents and every referenced value is required to be versioned, the values need to be valid extent IDs and thus have to be of the same data-type as extent IDs.

Goby's key/object store uses a two level architecture. The main tree maps keys to objects. Each object is in turn represented by a tree mapping offsets to data. Therefore, the `hashtree` and `rangetree` flavours are predefined for these use-cases. The key size of `hashtree` can be chosen freely by a preprocessor definition, whereas `rangetree` uses `uint64_t` for keys.

4.2.1 Memory Layout

Both nodes and leaves use the internal layout shown in figure 4.3: An extent is filled with N keys of size S_K and $N+1$ values of size S_V. If the extent size S_E is not of the form $S_E = S_V + x * (S_K + S_V)$ there will be a padding of $(S_E - S_V) \bmod (S_V + S_K)$ bytes. `0xFF..FF` is used as a NULL value marking empty keys and values.

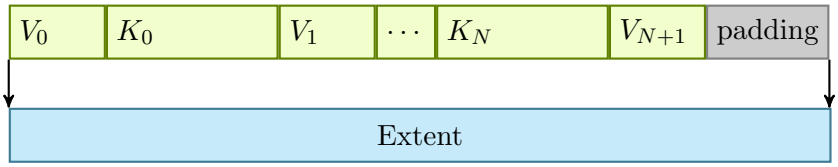


Figure 4.3: Tree node to extent mapping

Leaves store the value for the key K_i in V_{i+1}, leaving V_0 empty. Due to the properties of a B+Tree, any key K_i in an internal node of the tree has a left child with smaller keys and a right child with keys greater or equal. As a consequence, if a key K_i is not empty, value V_{i+1} is guaranteed to point to a valid extent. Also, if a key K_i is empty, all keys K_j and values V_{j+1} for $j \geq i$ will be empty.

4.2.2 Key Lookup

The function `find_leaf` is used whenever an operation needs to find a certain key in the tree. In the trivial case, this function will simply descend along the proper child pointers and return a stack containing the nodes traversed along the path to the leaf the key belongs into, including the leaf itself. This implies that the returned stack will always have at least one entry. The leaf potentially containing the key that was searched for is always on top of the stack.

In order to handle operations which modify the tree such as insertions and deletions, `find_leaf` takes an operation hint as well as a desired version. The operation hint tells `find_leaf` what kind of operation will be done to the found leaf, whereas the version specifies the desired version of the leaf. In case of an operation hint for a modifying operation, `find_leaf` will shadow any nodes with versions that are lower than the desired version on its path towards the leaf.

To keep the balancing invariants of a B+Tree, `find_leaf` will perform eager joins and splits if it detects that the next child node or leaf will violate an invariant if the operation indicated by the hint is executed. Hinting at an insertion will therefore cause full nodes to be split and deletion will join nodes or move entries between siblings. Shadowing as well as rebalancing are only performed on nodes below the root, since modification of the root will require references to it to be updated. Keeping the root within invariants is thus the responsibility of the caller.

Although eager joining and splitting might result in unnecessary overhead in special cases such as overwriting an existing key, the complexity arising from handling special cases in lazy variants is a potential source of bugs and overheads not worth the marginal benefit in very specific situations.

See algorithm 4.3 for an overview of `find_leaf`'s behaviour. Shadowing is performed whenever a node is encountered with a version lower than the one given to `find_leaf` and the operation hint is for a non-read operation. If the hint indicates a *write* and the next node is full, the node is split and the two halves inserted in the current node. Since we never descend into a full node with a write hint, there is always room to store

Algorithm 4.3 Overview of `find_leaf`

```
push current node onto stack
n ← get value for key
if version(n) < desired_version ∧ hint ≠ read then
    n ← shadow(n)
    version(n) ← desired_version
end if
if full(n) ∧ hint = write then
    n, n′ ← split(n)
    insert n′ into current node
end if
if half_full(n) ∧ hint = delete then
    join with adjacent, half full node or move key over
end if
descend to n
```

the second half. Encountering a half-filled node with a hint of *delete* will either join the node with an adjacent node if possible (ie. if an adjacent node also is only filled half) or alternatively move a key from an adjacent node to the current one. The participating node in these cases will also have to be shadowed if it is of an older version. Deletion will therefore never produce an empty node, since we never descend into a node with less than half entries.

4.2.3 Traversal

Traversal is done in a two step process. `traversal_init` is called first to build a handle containing the current position in the leaf as well as the stack for this leaf. This is achieved by calling `find_leaf` on the key the traversal starts at. If the exact key does not exist, `next_sibling` is called to find the next higher key in the tree. Should there be no more keys or the found key is bigger than the traversal's end key, an error is returned. The second step is to use `traversal_next` which returns the current key/value pair and calls `next_sibling`. Should there be no more keys satisfying the traversal bound, the handle is freed and the traversal's end signalled by setting `finished = true`.

The function `next_sibling` takes a traversal handle and advances it to the next higher key in the tree. If the last key in the current node is encountered, the parent node is retrieved from the stack and we descend into the next leaf to continue at its first key. If every entry in the parent has also been visited, this process continues recursively up to the root node of the tree.

4.2.4 Merging Versions

Goby contains two different merging algorithms: `merge_with_log` and `merge_fast`. Since the fast variant does not work for all situations it can be used optionally and in

case it fails, the merge will by retried with the log-assisted variant. Both merge implementations merge the changes of the `top` version onto the `bottom` version, serializing the order of operations on the tree as if the changes from `bottom` were applied first and the changes from `top` afterwards.

`merge_with_log`

The implementation of `merge_with_log` requires an operation log in the form of a set of key/value pairs that have been inserted into the tree to produce the `top` version. It will start by setting the result to be the `bottom` version and then reapply the operations from the log. An increased version number is used in order to correctly shadow nodes that are changed while replaying the log. Deletions would have to be recorded accordingly. Since Goby currently does not contain the complete code necessary for deletions on the transactional layer and above, they are not yet handled in the merge.

Figure 4.4 shows this process. Both versions modify one part of the tree, B $\rightarrow$ B' for the version on the left-hand side and C $\rightarrow$ C' for the version on the right-hand side. To merge the right-hand side onto the other, we start off by taking the result of the left-hand side modification and then redo the write to C, producing a new node C" and a new root A" shadowing the old tree.

`merge_fast`

Having a common ancestor as well as the `bottom` and the `top` versions satisfying the requirements outlined in section 3.2 allows us to perform a more efficient merge.

Figure 4.5 shows the basic principle: Both versions modify one part of the tree, B $\rightarrow$ B' for the version on the left-hand side and C $\rightarrow$ C' for the other one. Contrary to the log-based merge shown in figure 4.4, we don't create a new node C" by applying the log but directly take the already constructed node C' and insert it in the newly created parent node A".

However, since this algorithm has several tricky corner-cases when merging trees that underwent balancing, Goby's implementation will abort if B+Tree nodes with changes to the separating keys and therefore changes in overall structure are encountered. This is done in a two-pass process, where `merge_fast_check` is called to recursively compare the top and bottom tree to the common ancestor. If the tree structure allows for a simple fast merge, `fast_merge_execute` is called to perform the merge, using `fast_merge_reversion` to adjust version numbers in order to preserve versioning requirements: The new version number that was used as a version for newly created extents in the top tree has to be updated to reflect the merge. This can be achieved by replacing every instance of this number with the version of the bottom tree incremented by one. This gives local modifications in the top tree the correct version number which is relative to the bottom tree and will serve as the ancestor in further merges.

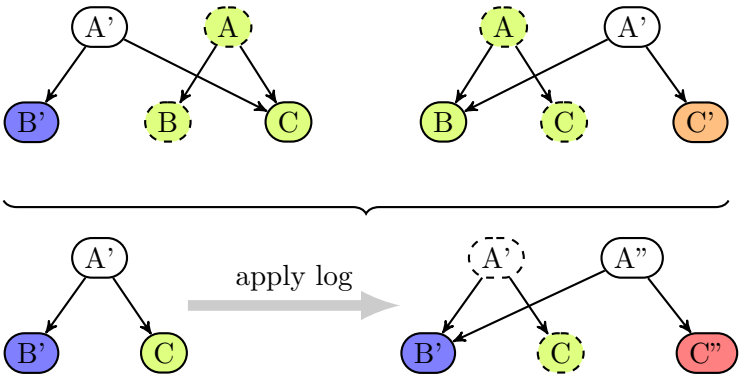


Figure 4.4: Merging trees by replaying logs

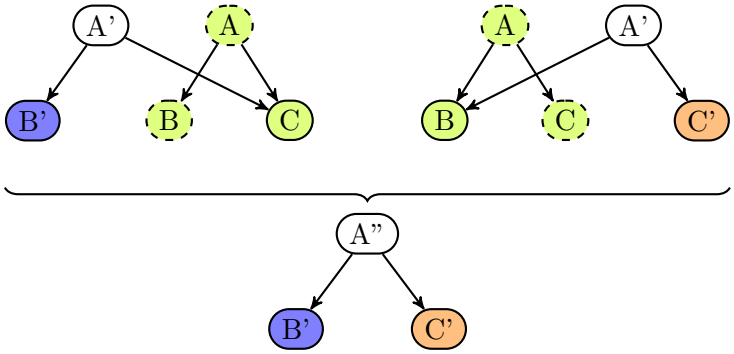


Figure 4.5: Merging trees without replaying logs

4.3 Transactions

Transactions are an extension over the versioned B+Tree, adding a shared, blessed version. Once a transaction is started, a reference to the currently blessed version is acquired. Any write operations will result in a newer version based on the acquired version, shadowing the original. Committing acquires another reference to the currently blessed version. Should this version differ from the one the transaction is based on, we check for conflicts by examining the read log and merge our local tree onto the new blessed version. After the merge, the resulting tree is promoted to be the new blessed version.

Promotion comes in two flavours. The simple approach is to acquire a lock while merging to prevent other processes to concurrently change the blessed version. The lock is released after the blessed version has been updated. Another approach is an iterative process of merging with the blessed version and then using the atomic `compare_and_swap` instruction to update the blessed version. This can of course fail due to concurrent modification and we have to repeat the process. Goby internally refers to this approach as *fluid committing*, this thesis also uses the term *iterative committing* for this principle.

Since conflict detection for read operations can only be done with a complete log of all read operations, the transactional layer has to keep a read log, logging all operations that read from the tree the transaction is based on. Additionally, the log-assisted merge algorithm requires a log of all write operations, therefore Goby also keeps a write log.

4.3.1 Logging

Goby can be compiled to either use CCAN's[1] `darray` to store operations or a custom hash set implementation. The hash set reduces the complexity of membership tests from linear to amortized constant, but does not keep the order the operations have been applied. Since Goby is oblivious to the order, using the hash set is the default since it is faster for insertion and lookup. The set is grown in powers of two to avoid extensive `realloc` calls. Another advantage of power of two size is that quadratic probing in the form of $P_k = P_{k-1} + k$ is guaranteed to find a spot if the set is not full.

Read operations are logged if the version of the extent referenced by the key we are reading from is older then the new version the transaction is producing. This prevents the logging of read operations of data that has been written in the current transaction. Write operations are always logged. Using the hash set will also prevent repeated reads and writes of the same key to generate redundant entries in the log.

4.3.2 Nested Transactions

Since a key stored in the B+Tree can point to arbitrary extents, we can construct multi-level hierarchies of trees. This can be used for example to construct directories: The first-level tree maps names to trees of the second level, whereas the second-level tree maps offsets to data extents. For such hierarchies, it is beneficial to have a notion of nested transactions, if the lookup in the first level and the modification in the second

[1] The Comprehensive C Archive Network, `http://ccodearchive.net`

should be part of the same transaction where merging the changes in the nested tree is desired. The transactional layer contains the function `nested_transaction` which will start a nested transaction on the tree referenced by the given key. Should the parent transaction be committed, any nested transactions will also be committed recursively.

Whenever the root of a nested tree covered by a nested transaction changes, the entry in the parent tree has to be updated to reflect this change. However, when merging the parent tree with another version extra care has to be taken. Since both merge algorithms are oblivious to the concept of nested transactions, they will simply see a difference in the value of a key associated with a nested transaction and handle it as if it was overwritten. Because of this all changes the other process has performed on the nested tree will be lost. The commit algorithm therefore has to recursively commit nested transactions against the version supplied by the foreign process to incorporate both sets of changes.

4.3.3 Problems Encountered

Splitting and joining nodes on demand requires knowledge about the leaf an insertion resp. deletion will modify. Splitting and joining are only necessary if the leaf violates the tree properties after the operation has been performed. However, once one of these balancing measures is required, nodes on the path to the root can subsequently require balancing as well. One possibility is to handle such situations recursively, passing back information about performed balancing and created nodes. The recursive approach proved to be overly complicated in differentiating these conditions and thus also prone to bugs. Proactive splitting and joining as implemented in Goby avoids the complexity by ensuring that modifications and balancing on lower nodes will not require balancing of nodes closer to the root.

4.4 Key/Object Store

The key/object store uses a `hashtree` to map from keys to objects. Objects are stored in `rangetree`s mapping offsets to data extents. Figure 4.6 illustrates how the value of key C is the extent ID of the root of a `rangetree` that in turn maps byte ranges to extents containing the object's data.

Creating an empty object is as simple as creating a new `rangetree` and inserting its root extent into the `hashtree`. The extent ID of the `hashtree`'s blessed version is stored in the extent with ID 0. The key/object store's `init` function will create an empty `hashtree` with version 1 and store it as the blessed version.

Beginning a transaction on the key/object store starts a transaction on the `hashtree` that is the currently blessed version. Read and write operations start a nested transaction on the appropriate `rangetree` for the desired object. Committing directly translates to a commit on the `hashtree` transaction which will in turn also commit the nested transactions for any objects modified during the course of the transaction. Since Goby restricts writes to multiples of the extent size, writes produce or replace whole extents which translate to a key/value pair in the `rangetree`.

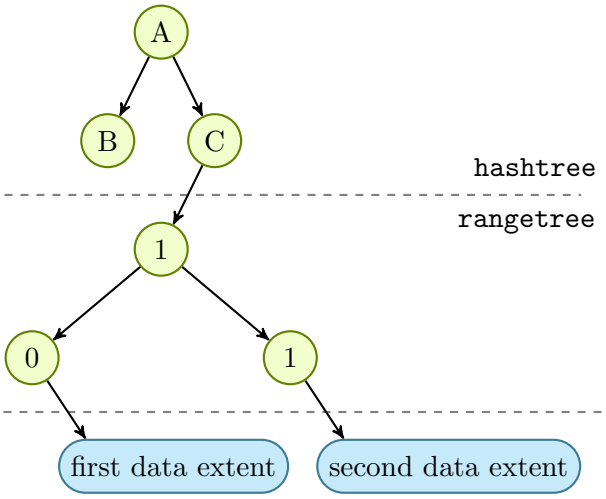


Figure 4.6: Key/object store to tree mapping

4.5 File System

Although Goby does not yet include a file system, we shall outline the possibilities for extending the implemented key/object store into a full file system.

4.5.1 Building Blocks

We have already seen how the payload of a file itself can be stored in the key/object store: A `rangetree` can be used to map byte ranges to data extents. We can also use a versioned tree for directories. However, since the tree implemented in Goby only supports fixed-length keys, we need to map file/directory names to the restricted key space. A suitable hash function can be used for this transformation.

To store permissions and other meta-data, we cannot simply use a hierarchy of versioned trees. Another intermediary data structure is necessary to hold the required meta-data and a reference to the versioned tree. This is the case with both directories and files. This data structure should also support chaining, if the hash function used in directories has a probability for collisions higher than we are willing to ignore.

4.5.2 Hierarchical Structure

There are to main choices on how to map the file system tree to the primitives mentioned in the previous section. We can either nest these primitives similar to the directory structure or construct a system using an identifying key stored in the top level tree for every file or directory. Figures 4.7 and 4.8 show the two possibilities. The approach with the index tree has several advantages. Since directories only store the index number of a file or sub-directory, transactions modifying the contents of a referenced file or directory do not have to update the meta-data structure of the parent directory. Hard links are

also possible, although some form of garbage collection mechanism has to be introduced to avoid orphaned files or directories.

When creating a new file or directory in the index-tree variant, one has to chose a unused index number. A suitable key can be found by walking the tree to find the first index number that does not yet exist in the tree; however, this can be an expensive operation depending on the structure of the tree. Since the tree is expected to be dense, an approach to alleviate this problem could be storing a bit in every internal node denoting if the subtree below a node is full. When searching for a free slot, only non-full subtrees have to be visited. Re-balancing can be disabled for most cases, since the order of insertions is linear.

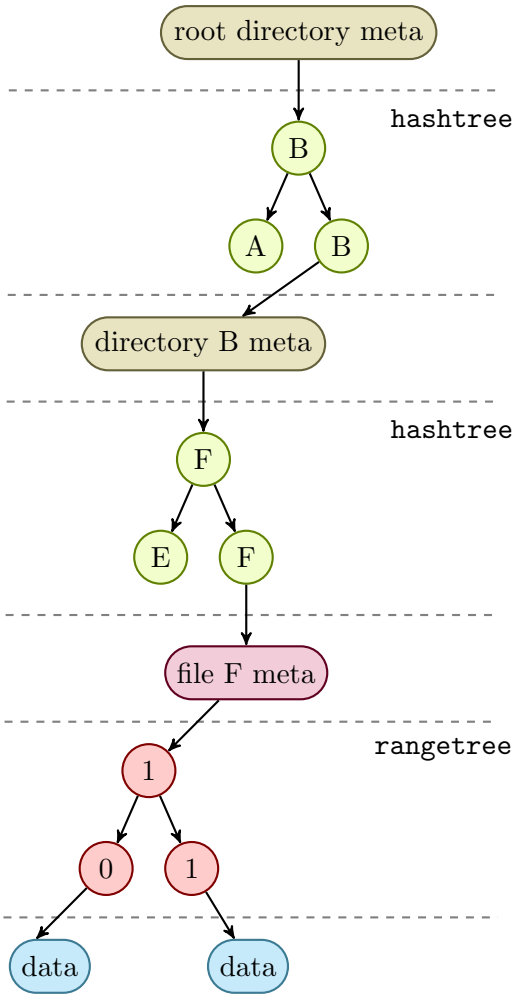


Figure 4.7: File system representation with nested structures.

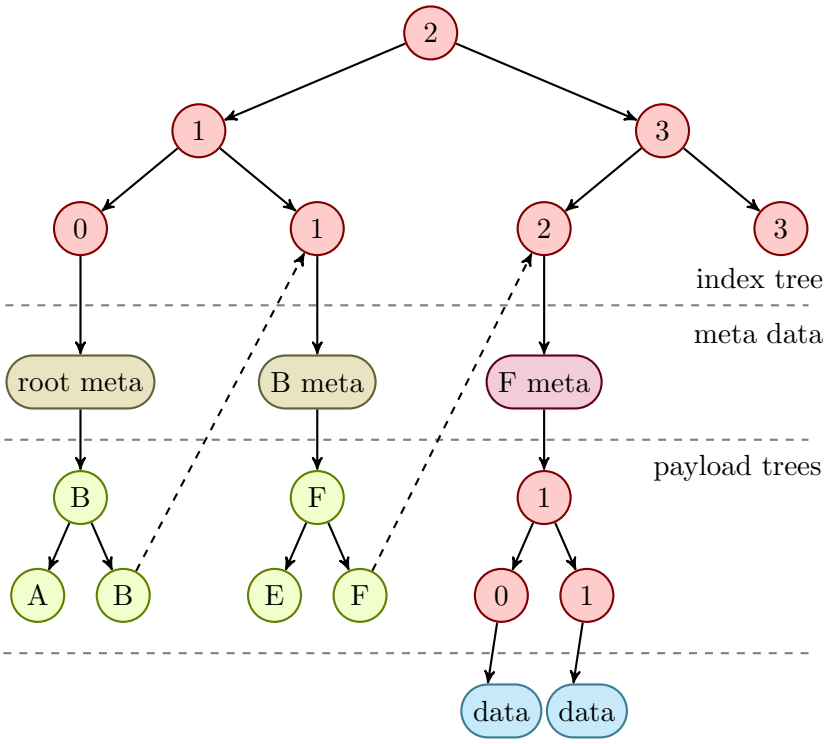


Figure 4.8: File system representation with an index tree. Directory entries reference the index number of sub-directories and files instead of direct nesting.

5 Evaluation

This section compares Goby's performance to that of `ramfs`, an in-memory file system used in the Linux kernel. Since Goby only contains a key/object store, we need to make abstractions and design workloads that are comparable for both systems under the limited set of operations Goby implements. Since Goby's goal is to be a stepping stone towards a usable file system, the focus of the evaluation is set on another file system.

5.1 Benchmarking Setup

The benchmarking program written for this thesis comes in two flavours. `workbench` is a program that maps a shared private region of memory into its address space and then forks into a number of processes executing the workload functions based on Goby. `nativebench` forks into a number of processes running the same workload functions translated to use the POSIX file API on a `ramfs` file system mounted in `/tmp/bench`. The child processes in both variants wait for a shared variable to be set to 1 before they run the workloads. Time is measured from setting this variable until all child processes have terminated.

A fabric `fabfile.py` is included for compiling and running the binaries on a remote host via SSH. Each configuration is run three times and averaged. All benchmarks have been run on a recent version of the Ubuntu Linux distribution. The used kernel version is 3.2.0. Significantly older versions such as 2.6.32 employed by the stable Debian distribution at the time of writing this thesis still have remnants of the big kernel lock in the VFS layer. An example is `llseek`, which uses a mutex leading to bad performance for `ramfs` in concurrent file seek calls. The two machines used are shown in in table 5.1.

The graphs shown in this section have the displayed range chosen such that the difference in performance characteristics between `ramfs` and Goby is visible. Since gottardo has the faster memory system and bigger caches, this can lead to differences in the ranges covered by the y-axes of graphs shown side-by-side.

5.2 Raw Write Throughput

Determining concurrent write performance can be done by running N processes ($P_0 = 0, \ldots, P_x = x, \ldots, P_N = N$) performing I writes of B bytes to a file. We can implement the same workload on a POSIX API to get a comparison with any current Linux file-system. Listing 5.1 shows the two implementations of this workload. Note that the total

Name	gottardo	gruyere
Architecture	Intel Beckton	AMD Barcelona
Cores	4x8x1 (HT disabled)	8x4x1
L1 cache	384 KB	4 x 64 KB
L2 cache	2 MB	4 x 512 KB
L3 cache	24 MB	2 MB
RAM	128 GB	14 GB

Table 5.1: Machines used for benchmarking

Algorithm 5.1 POSIX

```
1: for 1 → I do
2:     fd ← open(filename)
3:     lseek(fd, position)
4:     write(fd, data, B)
5:     close(fd)
6: end for
```

Algorithm 5.2 Goby

```
1: for 1 → I do
2:     tr ← begin()
3:
4:     write(tr, filename, data, position, B)
5:     commit(tr)
6: end for
```

Figure 5.1: Writer implementation for benchmarking. The position is always 0 except in disjoint writing mode, where it is $P_i * B$. The filename is either "file0" for writing to the same file or "file" + P_i for writing to different files

bytes written is given by $N * I * B$. Goby's results presented in this section have been performed with *private allocation* mode, iterative committing and fast merges enabled.

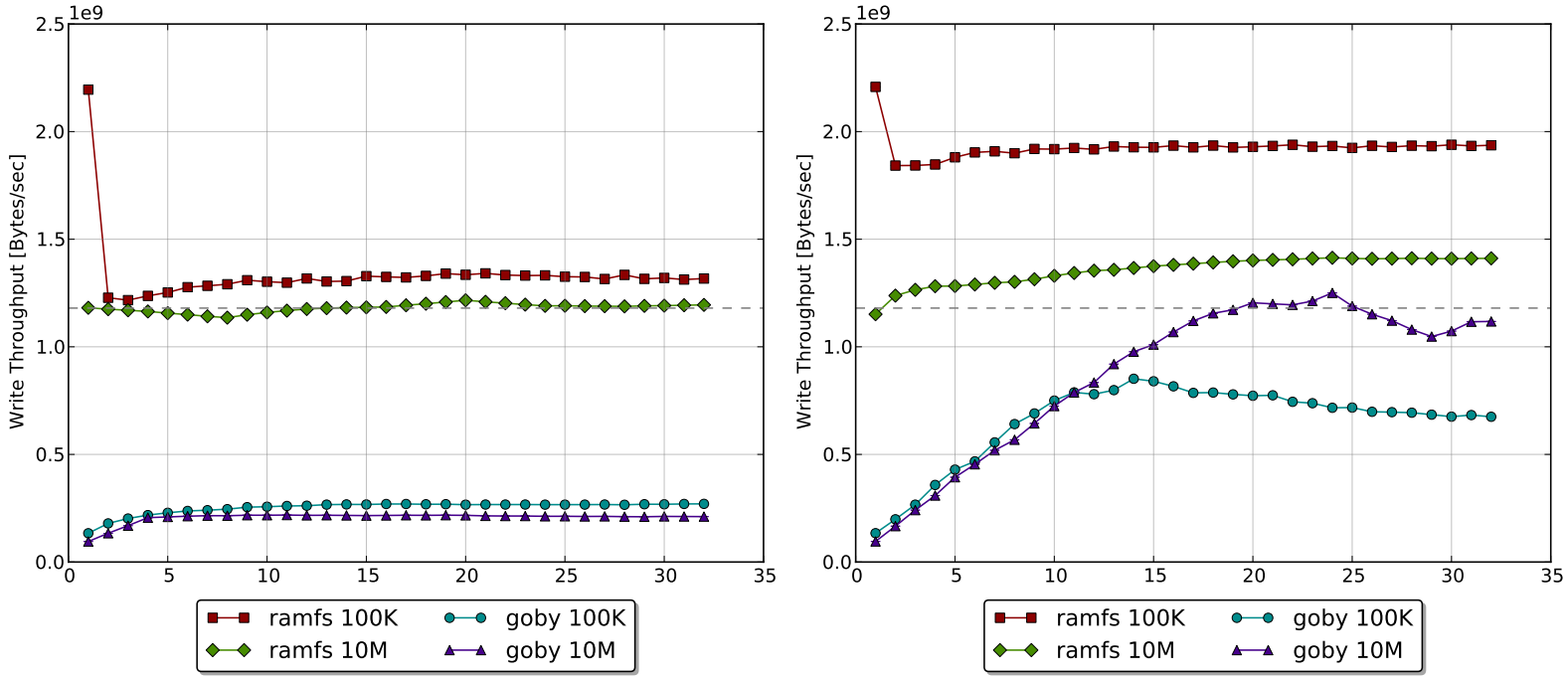
5.2.1 Writing to the same File

Figures 5.2 and 5.3 show the throughput for the workloads operating on a shared file. The parameters used are $B = 102400$ and $I = 10000$ for the 100K variant and $B = 10240000$ and $I = 1000$ for the 10M variant.

Performance is mainly dominated by the memory system, where Goby is constantly slower than `ramfs` due to overhead in merging and worse cache utilization. The jumps in `ramfs` performance are most likely due to side-effects of caching.

Both of Goby's merge algorithms are not efficient in handling writes to the same region, since the log-assisted merge has to replay the log and `merge_fast` has to walk every key stored in the tree. However, when writing to disjoint regions, `merge_fast` leads to an increase in performance for larger writes, since it can reuse entire subtrees. There is still a performance penalty due to having to reversion these subtrees as outlined in section 4.2.4.

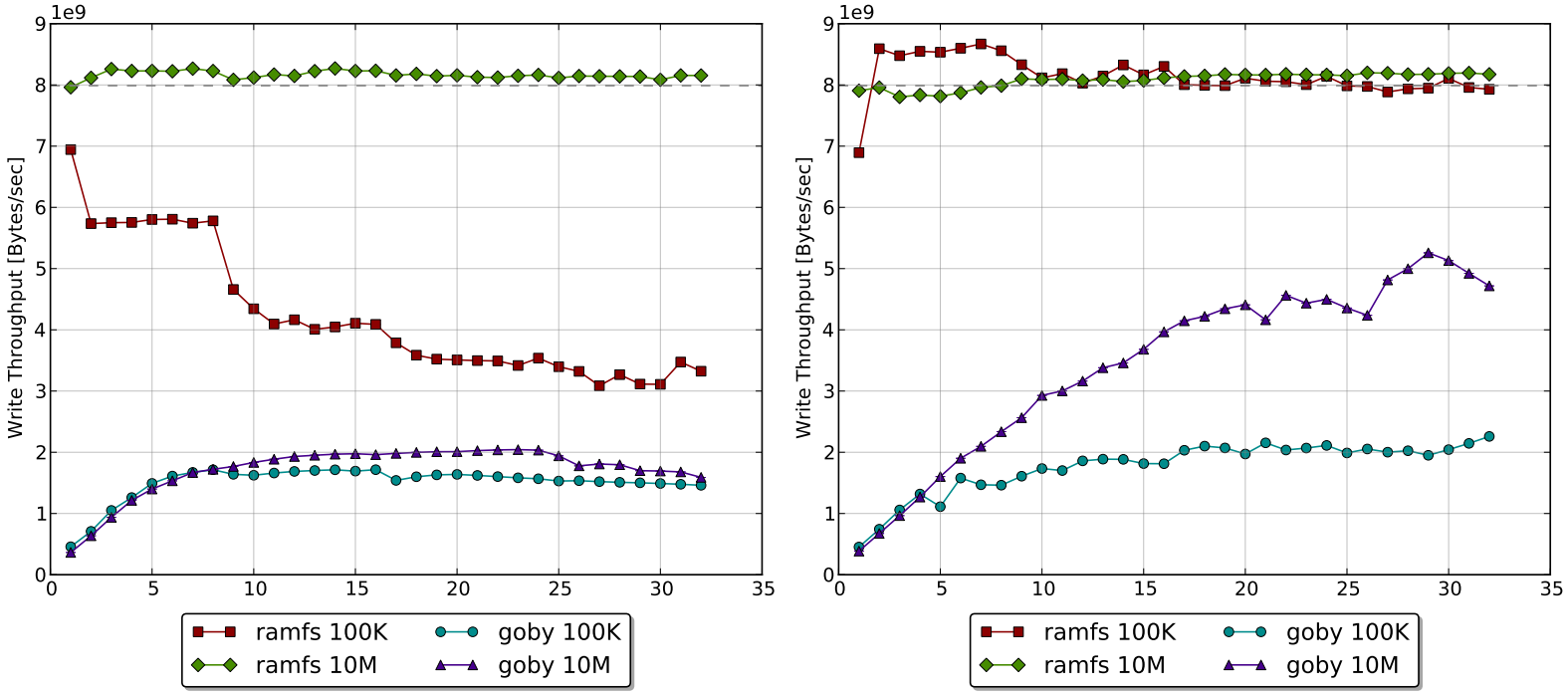
Smaller write sizes lead to contention in both the locks employed in `ramfs` as well as in updating the blessed version in Goby. Due to the extreme running times of these benchmarks, only partial results are available for the total number of bytes written used in this evaluation and are therefore not included.



(a) Same file, single shared region

(b) Same file, disjoint regions

Figure 5.2: Average write throughput on gruyere. The x-axes show number of concurrent processes while the y-axes show absolute throughput in bytes per second. The horizontal dashed line denotes single-process `memcpy` throughput as a reference.



(a) Same file, single shared region

(b) Same file, disjoint regions

Figure 5.3: Average write throughput on gottardo. The x-axes show number of concurrent processes while the y-axes show absolute throughput in bytes per second. The horizontal dashed line denotes single-process `memcpy` throughput as a reference.

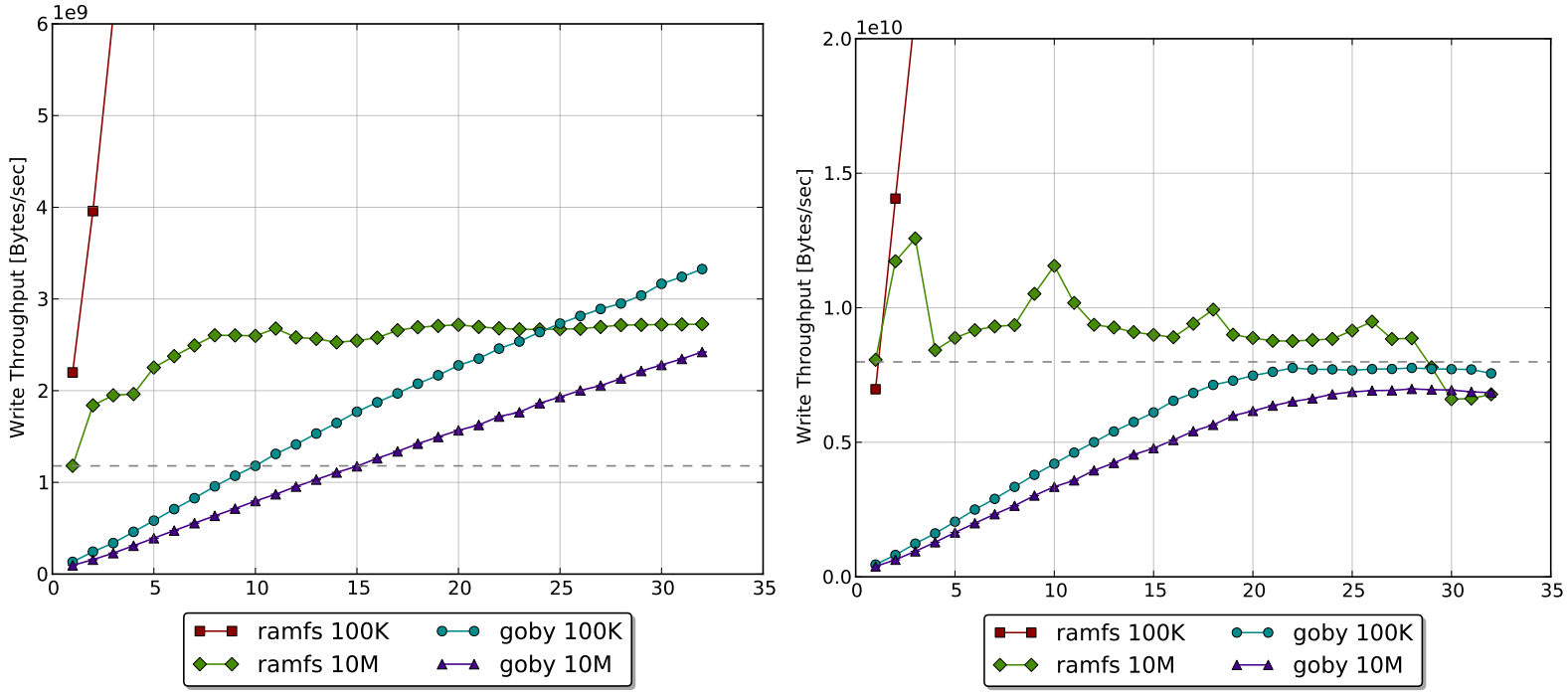
5.2.2 Writing to different Files

Figure 5.4 shows the throughput for the workload on different files. Each process writes to its own file. The parameters for the 100K and 10M variants are the same as described in the previous section.

Since the workload does not change the size of the files as well as access and modification times being disabled, `ramfs` does not have to perform updates of meta-data. Processes can apply writes to the address space containing the file's data without concurrent access from other processes.

In the 100K variant, the cache-friendly behaviour of `ramfs` leads to excellent results since the complete file fits into the CPU's cache. To focus on the comparison for the 10M variant the graph has been cut-off at a suitable value. However, the curve for the 100K variant of `ramfs` performance rises linearly in the unrestricted view.

In the 10M variant, Goby starts off with lower throughput due to the overheads of merging and unoptimized code but scales with the number of concurrent processes until it reaches the limits of the memory system. `ramfs` performance benefits from the bigger caches on gottardo whenever a new CPU is used (every 8 cores) but degrades again once a more than two processes are scheduled on the CPU. Since the processes are all writing to a different file and therefore also a different address space, more processes scheduled on the same CPU lead to more cache thrashing.



(a) Different files on gruyere

(b) Different files on gottardo

Figure 5.4: Average write throughput for different files. The x-axes show number of concurrent processes while the y-axes show absolute throughput in bytes per second. The horizontal dashed line denotes single-process `memcpy` throughput as a reference.

5.3 Raw Read Throughput with a concurrent Writer

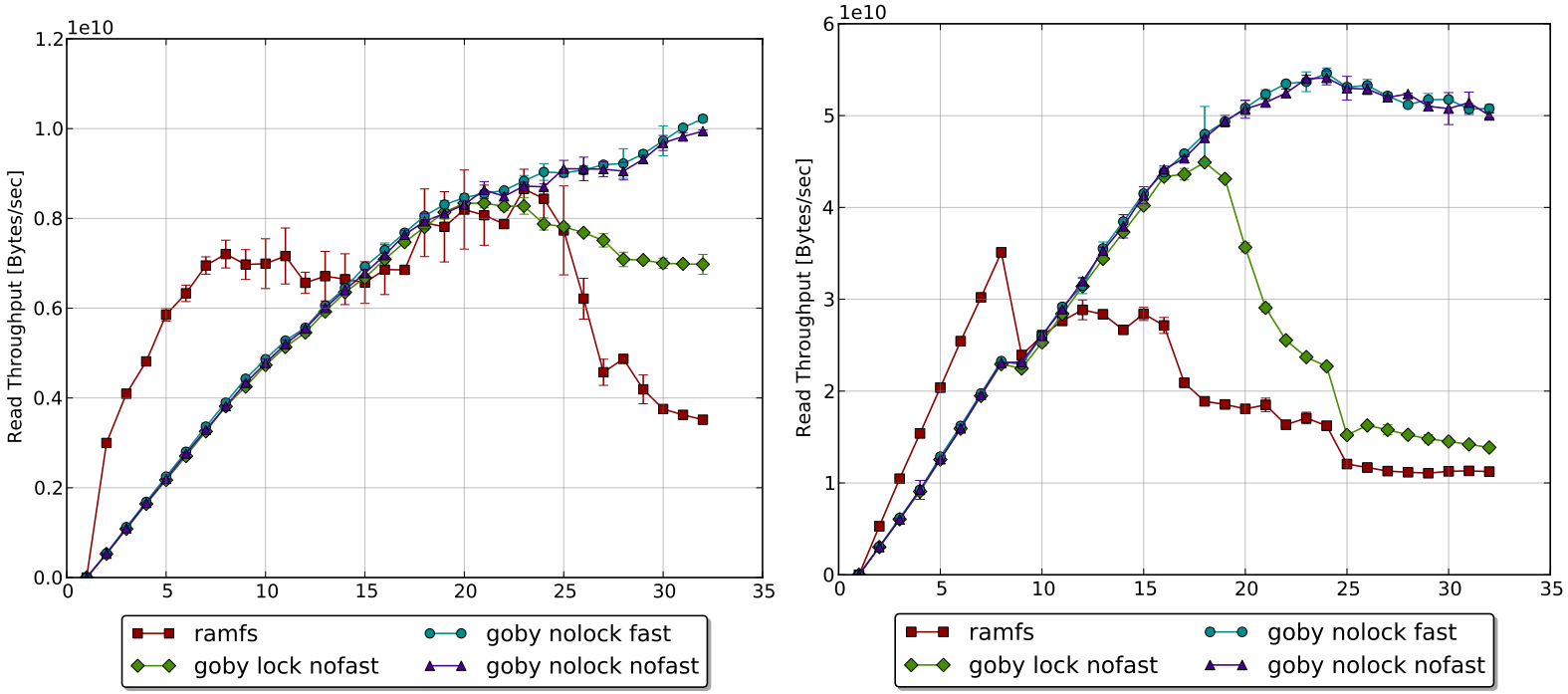
To look at the impact concurrent readers have on overall performance, we can execute the writer workload from the previous section while having $N - 1$ concurrent processes reading the written data over and over again. The readers exit once they read the value I at the beginning of the data file while the writer stores the current number of its iterations there. Since Goby does not have to perform merges with only one concurrent writer, the resulting scheme is very similar to RCU in that it also creates a copy and updates the pointer after the changes have been applied. However, Goby does garbage collection via reference counts instead of waiting for the grace-period and can directly overwrite the blessed version when the changes are committed.

Figures 5.5 and 5.6 show read and write throughput for this workload with 100K read and write sizes while figures 5.7 and 5.8 show the same for 10M sizes. The number of write iterations is chosen exactly the same as in the concurrent write throughput benchmarks. Therefore, the overall number of bytes written is the same for both variants. The overall bytes read however depends on the throughput.

`ramfs` has a higher base throughput for writes. Write throughput however degrades to the same value Goby is holding indifferent to the number of concurrent readers. An explanation for this behaviour lies in the grace-period used in RCU and can be worse if inter-processor interrupts are used to force a quiescent state.

In terms of read performance, we can see that read performance in Goby scales better than `ramfs` for a large number of concurrent processes. The only exception is the lock-based commit making reads of the blessed pointer more costly due to the additional memory barriers issued by the atomic CPU instructions.

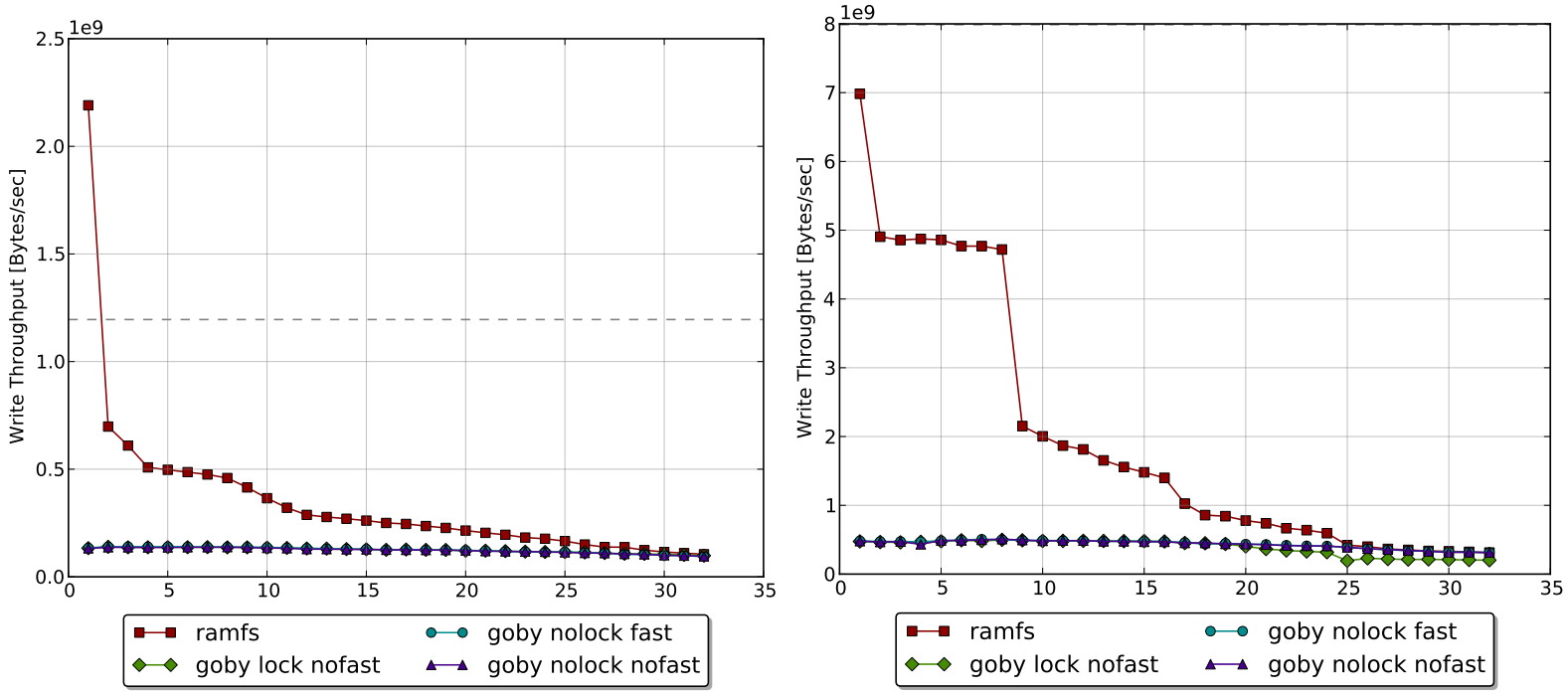
Since the 100K writes fit into the cache and the processes in Goby read from the most recent snapshot that is not concurrently modified, Goby is more cache friendly when repeatedly reading the same data. The larger 10M writes do not fit into caches, which brings performance of `ramfs` and Goby closer together.



(a) 100K on gruyere

(b) 100K on gottardo

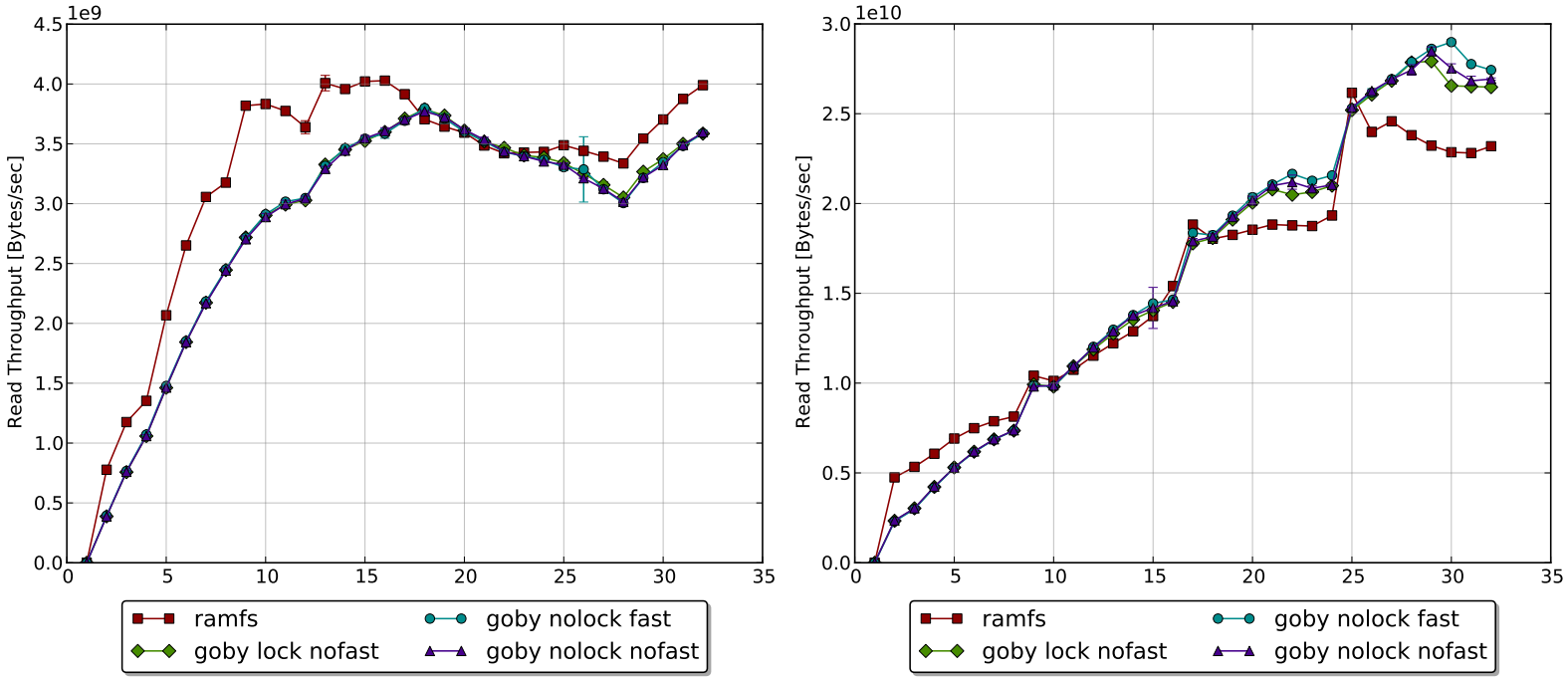
Figure 5.5: Read throughput with one writer and $x - 1$ concurrent readers. The x-axes show number of concurrent processes while the y-axes show absolute throughput in bytes per second. Operations are 100 KB in size.



(a) 100K on gruyere

(b) 100K on gottardo

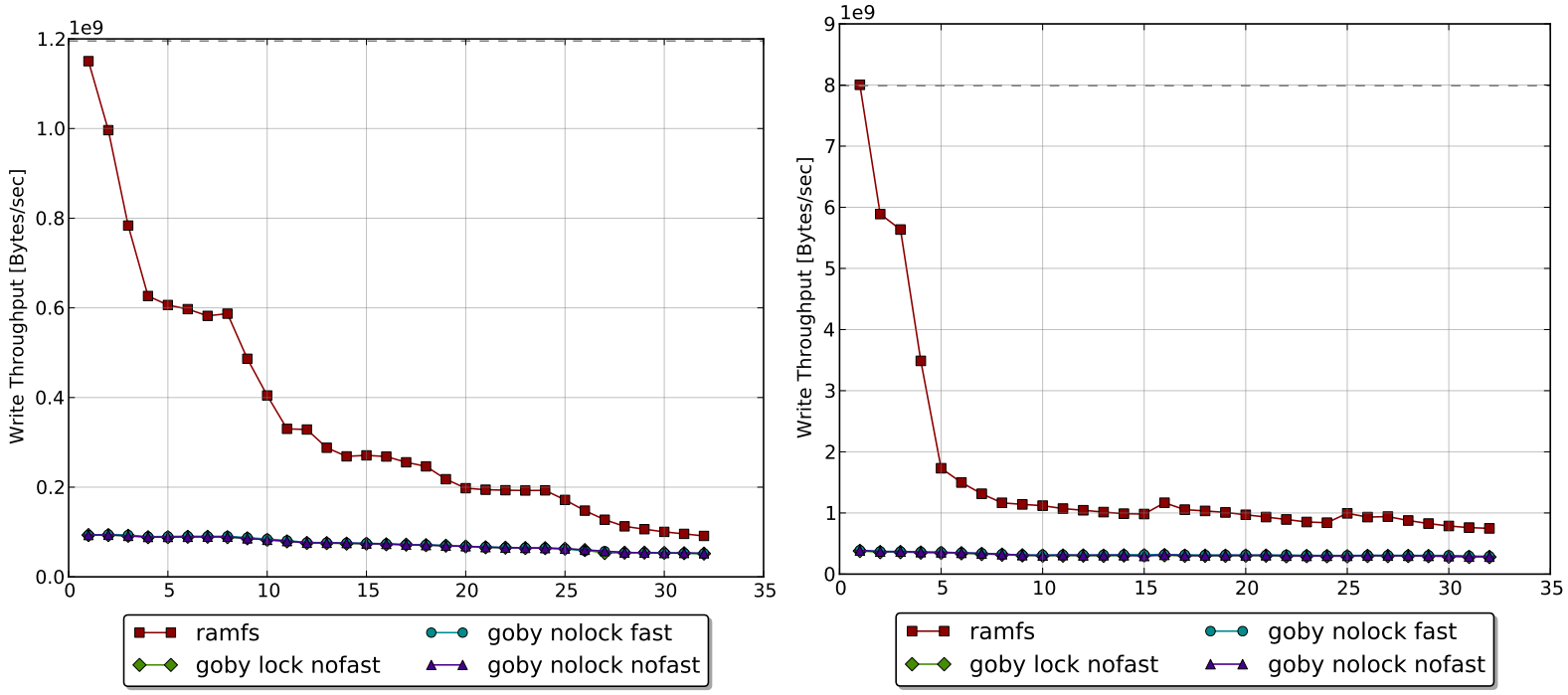
Figure 5.6: Write throughput with one writer and $x - 1$ concurrent readers. The x-axes show number of concurrent processes while the y-axes show absolute throughput in bytes per second. Operations are 100 KB in size. The horizontal dashed line denotes single-process `memcpy` throughput as a reference.



(a) 10M on gruyere

(b) 10M on gottardo

Figure 5.7: Read throughput with one writer and $x-1$ concurrent readers. The x-axes show number of concurrent processes while the y-axes show absolute throughput in bytes per second. Operations are 10 MB in size.



(a) 10M on gruyere

(b) 10M on gottardo

Figure 5.8: Write throughput with one writer and $x-1$ concurrent readers. The x-axes show number of concurrent processes while the y-axes show absolute throughput in bytes per second. Operations are 10 MB in size. The horizontal dashed line denotes single-process `memcpy` throughput as a reference.

5.4 Mixed Access Pattern

A more realistic access pattern can be achieved by mixing read and write operations in a pseudo-random pattern. Postmark [9] is a widely-used example of such a workload. However, since Goby does not yet support appending to objects nor deletion thereof, we need build a benchmark based on writing to pre-created files. Algorithm 5.3 describes the workload: During 10 seconds, a random sequence of read and write operations are performed. Parameters for the workload are number of files F and the read bias r in percent. The amount of data read and written is fixed to 100 KB per operation. Prior to running the workload, F files are created and initialized to contain 100 KB of data to prevent failing read operations on files that have not yet been written to.

Algorithm 5.3 Mixed access workload

```
while elapsed time < 10 seconds do
  f ← random file
  if rand() % 100 < read_bias then
    read from f
  else
    write to f
  end if
end while
```

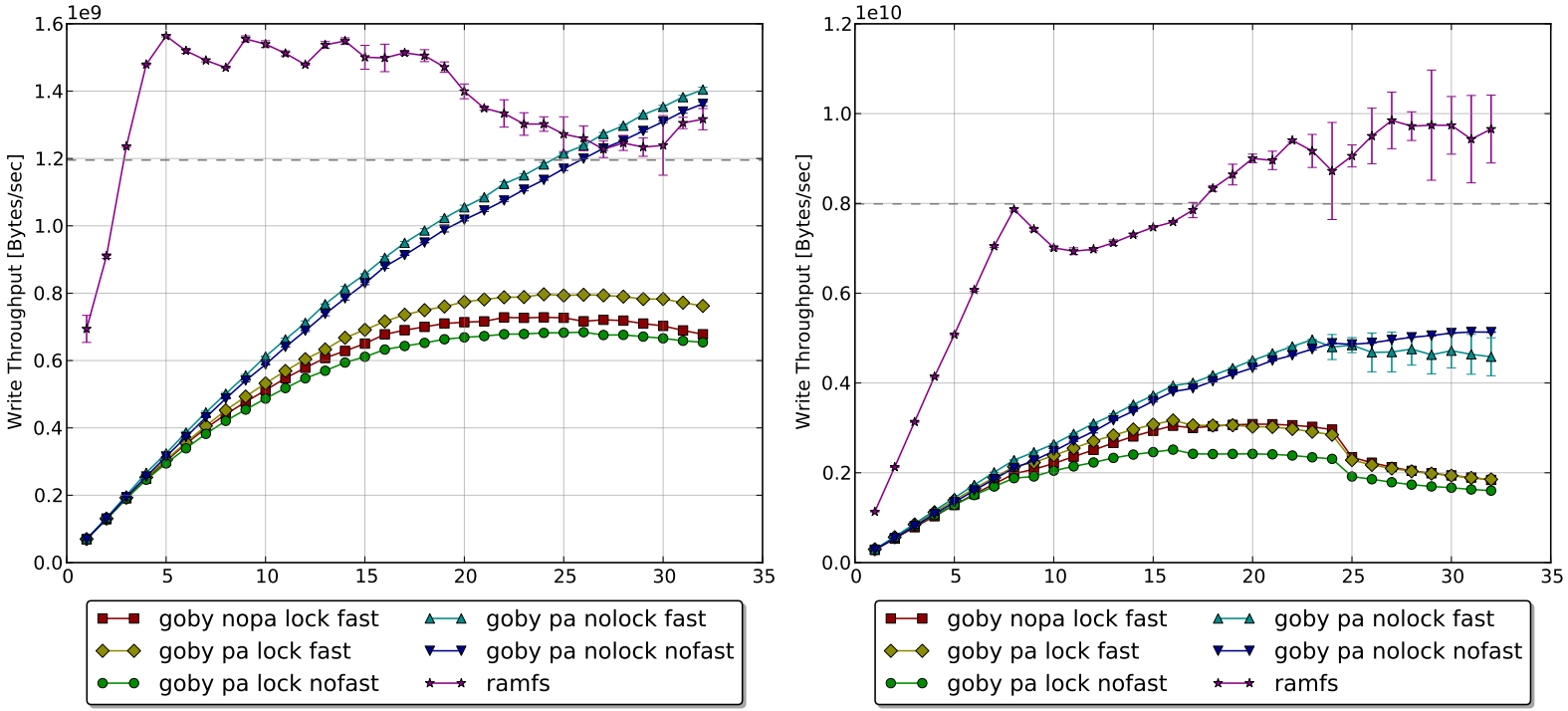
5.4.1 80% Read Bias

Figures 5.9 to 5.12 show the resulting write and read throughput for 80% reads and 20% writes. The former two use 10 files while the latter use 1000 files. Since writes are the limiting factor, read throughput is about equal to four times the write throughput in both Goby and `ramfs`.

In Goby, the three variants using lock-based committing scale significantly worse than the iterative approach. Using the private allocator further degrades performance with an increasing number of cores. Fast merge only offers little improvements in this workload since the different writers entirely overwrite each other, leading to a linear walk through all referenced data extents in fast merge, which is very similar to log-replaying in terms of complexity and access patterns.

`ramfs` starts off better than Goby but degrades similar to the lock-based variants of Goby with an increasing number of concurrent processes. Goby scales better but overall still suffers from worse cache usage which is even more visible on the bigger caches of gottardo.

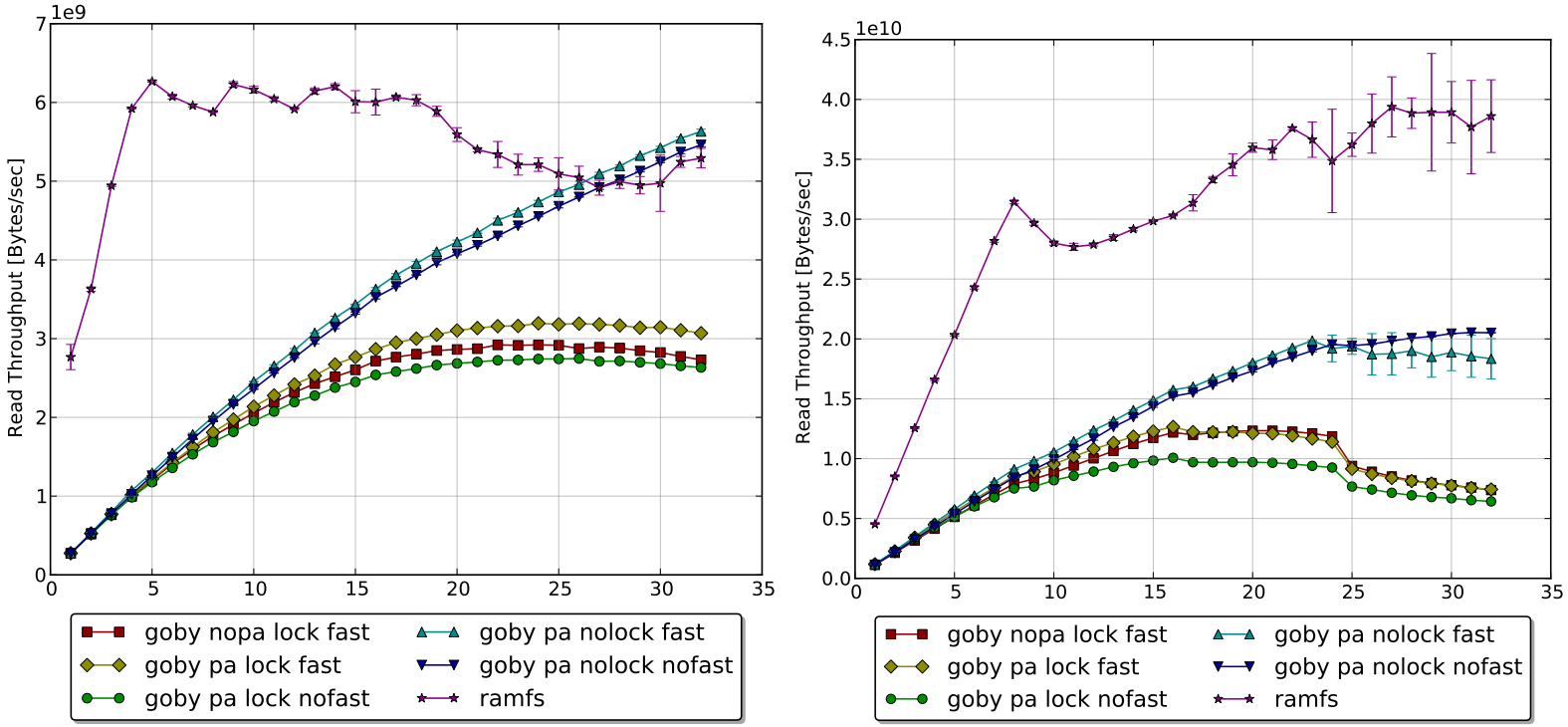
When using 1000 files, `ramfs` has an even bigger advantage over Goby. Similar results have been seen in section 5.2.2 and are also comparable, since an increase in the number of files leads to a higher chance to write to a file not accessed by a concurrent process. The effect is even more visible since operations are done with 100 KB sizes where `ramfs` holds a big advantage over Goby for access to independent files.



(a) Write throughput on gruyere

(b) Write throughput on gottardo

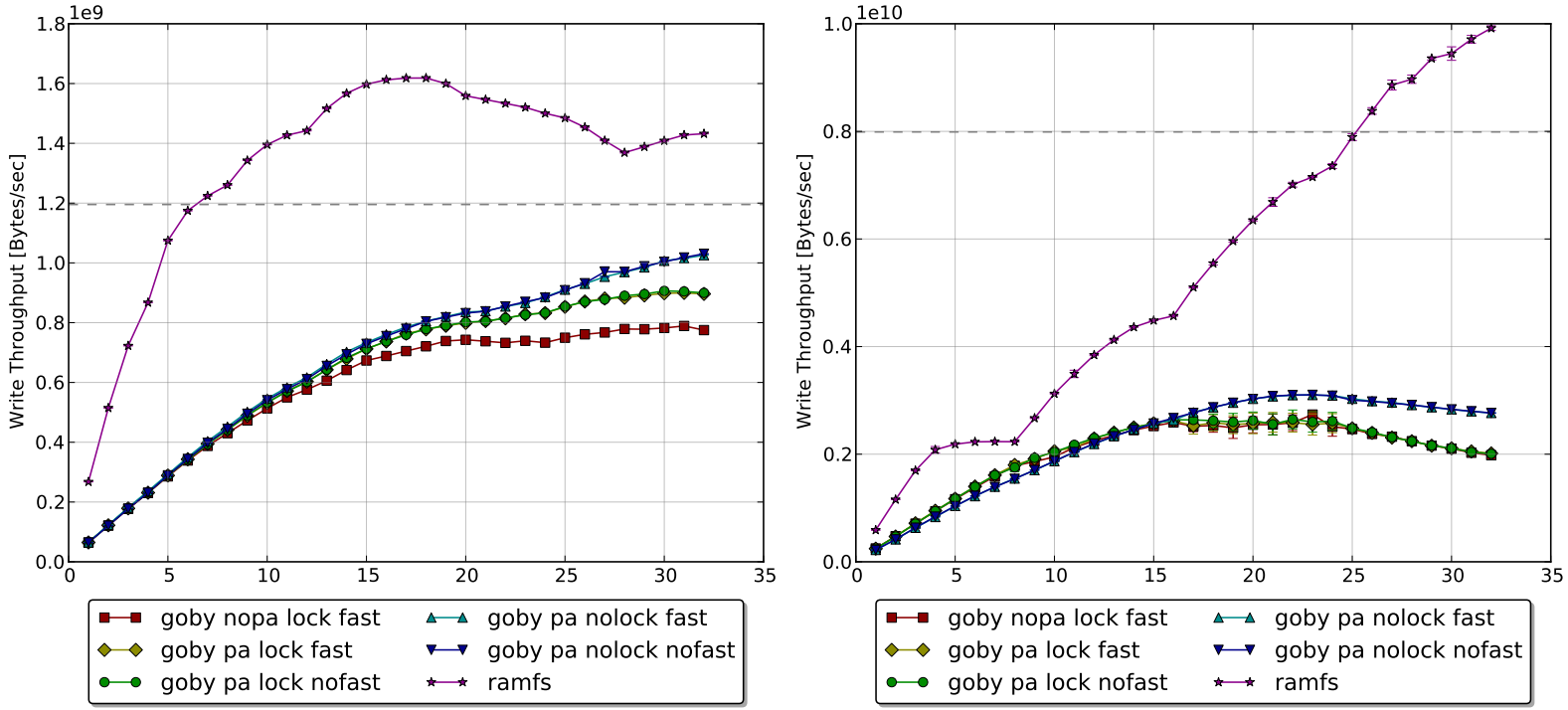
Figure 5.9: Write throughput for mixed access workloads. The x-axes show number of concurrent processes while the y-axes show absolute throughput in bytes per second. 10 files were used with a read bias of 80%. The different settings shown are pa/nopa for private allocation mode on/off, lock/nolock for locked and iterative committing and fast/nofast for fast merge on/off.



(a) Read throughput on gruyere

(b) Read throughput on gottardo

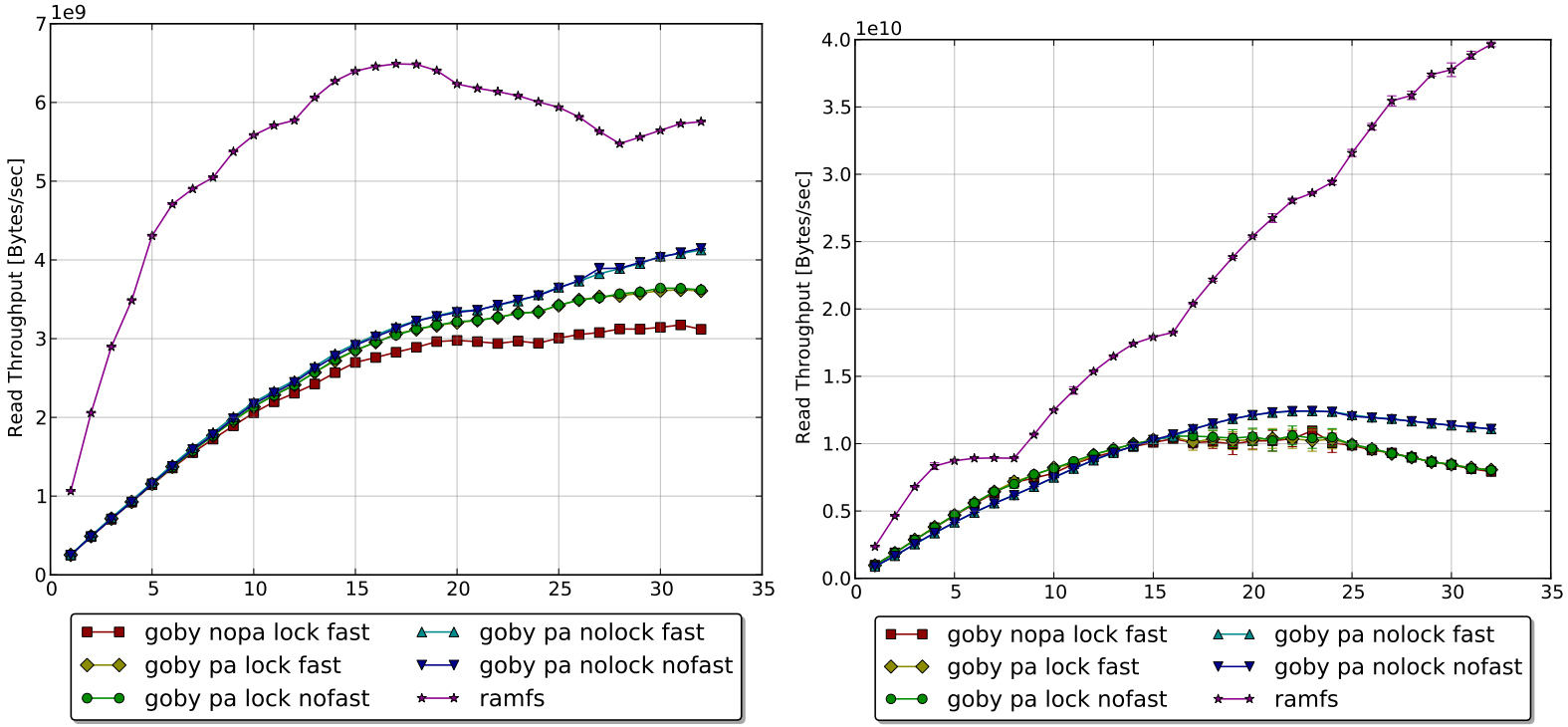
Figure 5.10: Read throughput for mixed access workloads. The x-axes show number of concurrent processes while the y-axes show absolute throughput in bytes per second. 10 files were used with a read bias of 80%. The different settings shown are pa/nopa for private allocation mode on/off, lock/nolock for locked and iterative committing and fast/nofast for fast merge on/off.



(a) Write throughput on gruyere

(b) Write throughput on gottardo

Figure 5.11: Write throughput for mixed access workloads. The x-axes show number of concurrent processes while the y-axes show absolute throughput in bytes per second. 1000 files were used with a read bias of 80%. The different settings shown are pa/nopa for private allocation mode on/off, lock/nolock for locked and iterative committing and fast/nofast for fast merge on/off.



(a) Read throughput on gruyere

(b) Read throughput on gottardo

Figure 5.12: Read throughput for mixed access workloads. The x-axes show number of concurrent processes while the y-axes show absolute throughput in bytes per second. 1000 files were used with a read bias of 80%. The different settings shown are pa/nopa for private allocation mode on/off, lock/nolock for locked and iterative committing and fast/nofast for fast merge on/off.

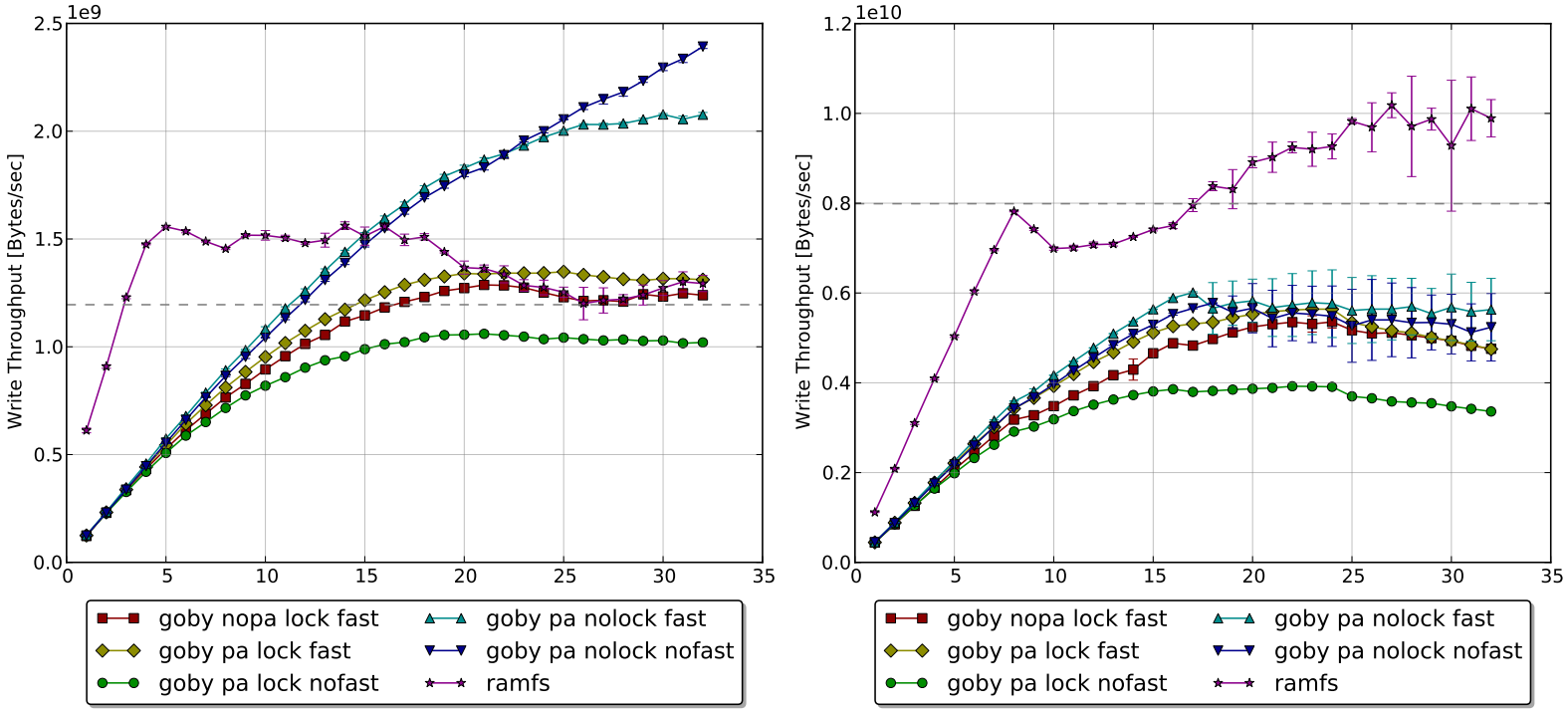
5.4.2 20% Read Bias

Figures 5.13 to 5.16 show the resulting write and read throughput for 20% reads and 80% writes. Again, the former two use 10 files while the latter use 1000 files.

As with the results for a read bias of 80%, the lock-based variants of Goby exhibit worse performance than the iterative approach to committing. However, we can see that on gottardo, the performance hit for some of the lock-based variants is not as strong anymore and results lie within the standard deviation of the iterative variants.

Write throughput of Goby manages to exceed that of `ramfs` for the smaller caches on gruyere and scales very similar when 1000 files are used, however hitting the limitations of the memory system on gottardo sooner due to higher overhead and worse cache utilization.

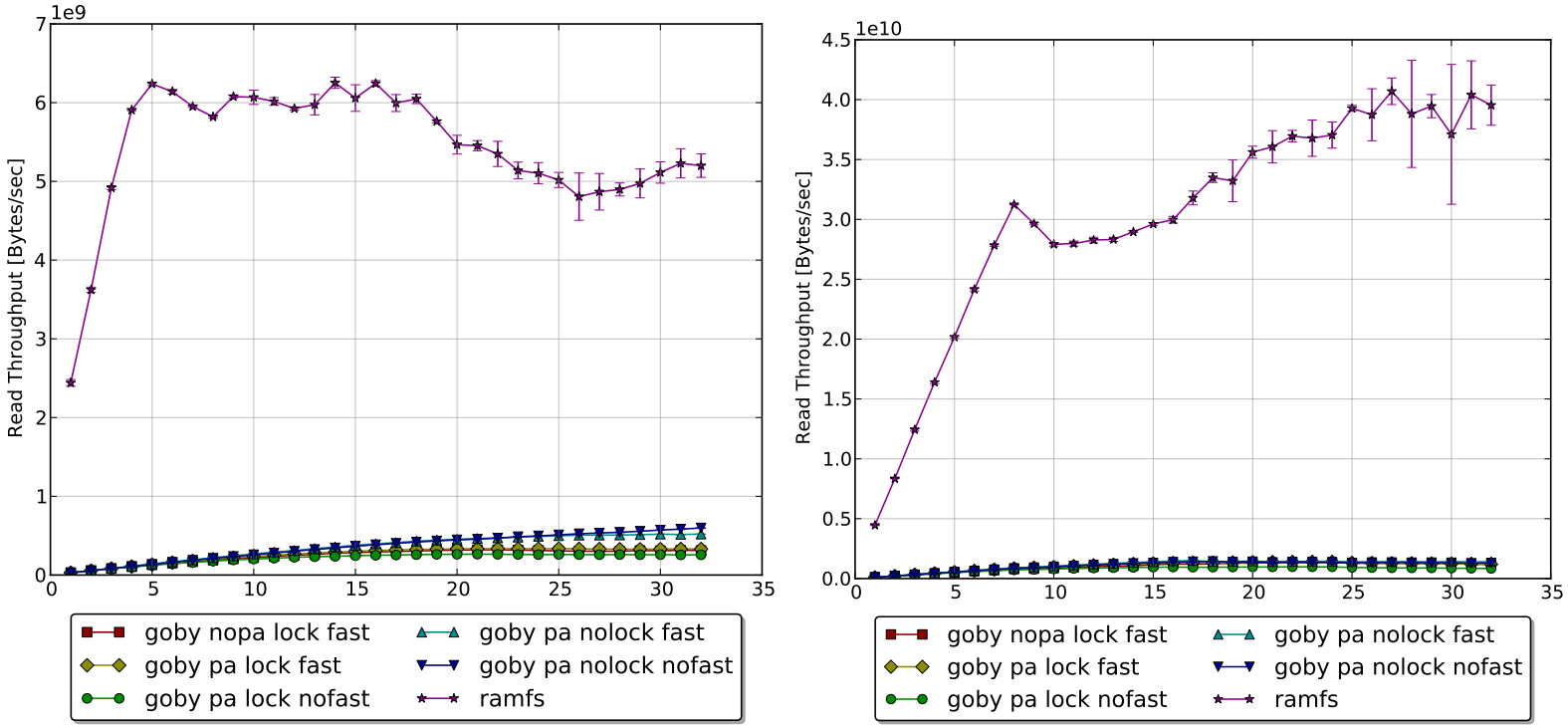
In all cases, `ramfs` outperforms Goby in read throughput. This can be explained by Goby's overhead in tree traversal and the extent size of 512 bytes compared to the page size of 4 KB. Looking at `perf` data reveals that a lot of time in Goby is spent in `next_sibling` and therefore also `traversal_next`.



(a) Write throughput on gruyere

(b) Write throughput on gottardo

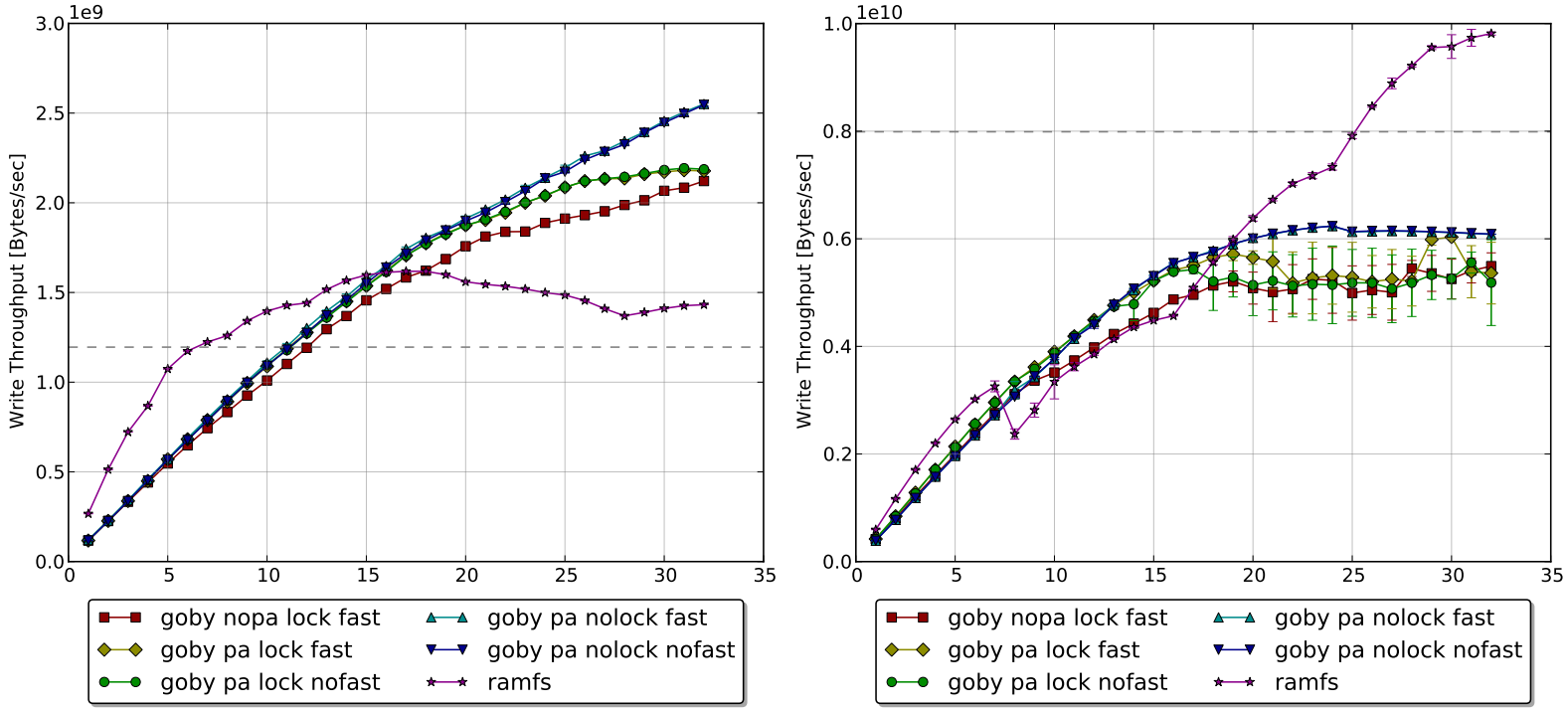
Figure 5.13: Write throughput for mixed access workloads. The x-axes show number of concurrent processes while the y-axes show absolute throughput in bytes per second. 10 files were used with a read bias of 20%. The different settings shown are pa/nopa for private allocation mode on/off, lock/nolock for locked and iterative committing and fast/nofast for fast merge on/off.



(a) Read throughput on gruyere

(b) Read throughput on gottardo

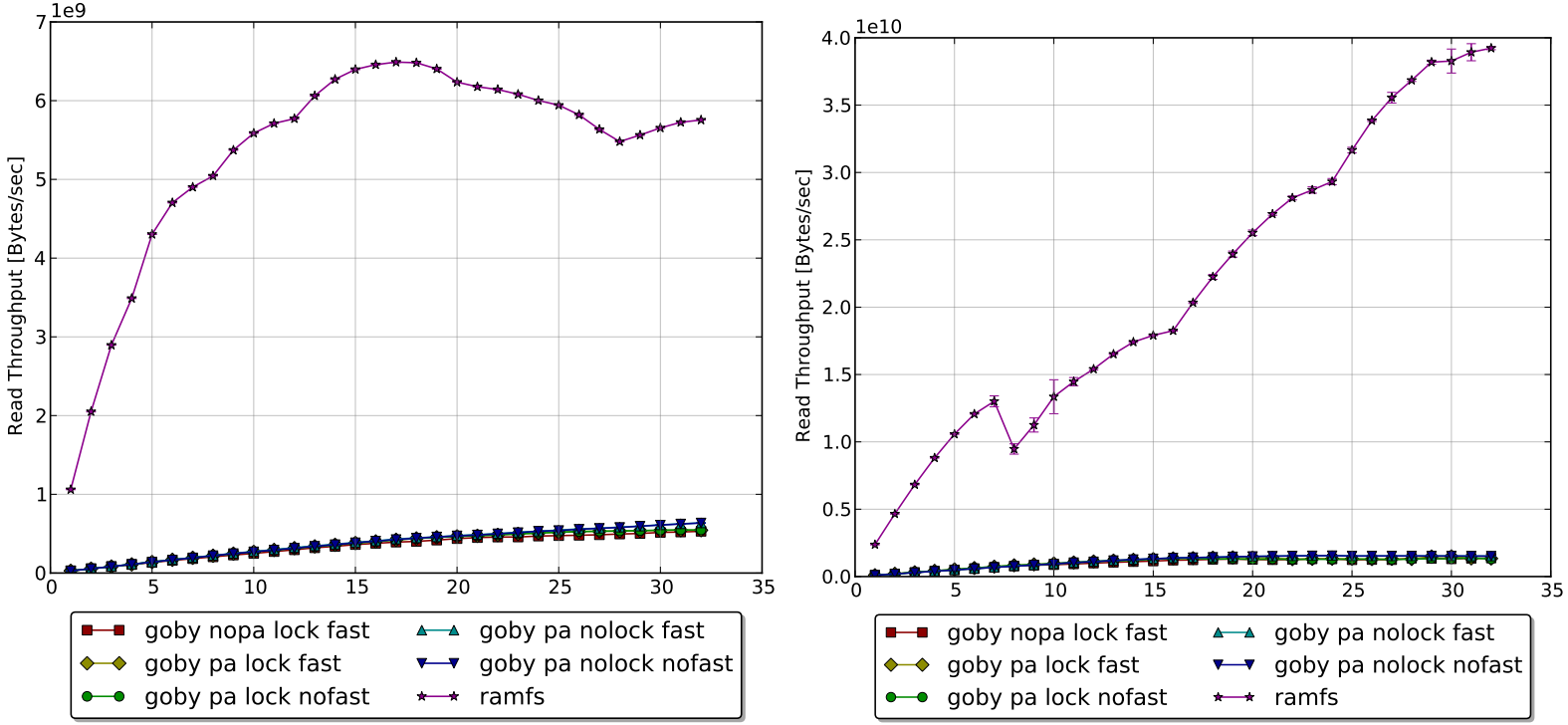
Figure 5.14: Read throughput for mixed access workloads. The x-axes show number of concurrent processes while the y-axes show absolute throughput in bytes per second. 10 files were used with a read bias of 20%. The different settings shown are pa/nopa for private allocation mode on/off, lock/nolock for locked and iterative committing and fast/nofast for fast merge on/off.



(a) Write throughput on gruyere

(b) Write throughput on gottardo

Figure 5.15: Write throughput for mixed access workloads. The x-axes show number of concurrent processes while the y-axes show absolute throughput in bytes per second. 1000 files were used with a read bias of 20%. The different settings shown are pa/nopa for private allocation mode on/off, lock/nolock for locked and iterative committing and fast/nofast for fast merge on/off.



(a) Read throughput on gruyere

(b) Read throughput on gottardo

Figure 5.16: Read throughput for mixed access workloads. The x-axes show number of concurrent processes while the y-axes show absolute throughput in bytes per second. 1000 files were used with a read bias of 20%. The different settings shown are pa/nopa for private allocation mode on/off, lock/nolock for locked and iterative committing and fast/nofast for fast merge on/off.

6 Conclusion

This thesis introduces Goby, a layered approach to implementing transactional data storage based on multi-versioning concurrency control. Goby's key/object store can be used as a model for file storage and extended into a full-blown file system supporting transactions. Writable snapshots are trivially possible by keeping multiple blessed versions of the root tree. Common semantics can be achieved by carefully choosing the duration of transactions. POSIX semantics can be achieved by wrapping every individual write call in its own transaction. If changes only have to be visible after closing a file, the transaction can be started when opening a file and committed upon closing which also functions as a way of batching operations with its inherent performance advantages.

The evaluation against `ramfs` as a widely used in-memory file system has shown that while outperforming it in certain areas, Goby has a lot of potential for improvement. Overall performance could be improved by better and friendlier memory access patterns. Workloads such as the concurrent modification of the same areas in shared files show the importance of a fast general-purpose merging algorithm. Log replaying is a simple approach and easy to implement. However, a smarter algorithm that exploits the versioning history and the structure of the underlying B+Trees can improve performance.

6.1 Future Work

Since this thesis is to be considered an intermediary step towards a scalable file system, there is a lot of potential for further refinements. This section lists a few possible extensions for this work.

6.1.1 Improving Performance

Better Merging

Since Goby writes changes to a local version of the tree, merging these changes back into the shared version is pure overhead. This hit to performance only makes sense if it is better than traditional locking and applying changes sequentially. This implies that the merge algorithm has to be faster than the write operations themselves.

Both of Goby's merging algorithms perform a lot of unnecessary operations in certain situations, for example when transactions overwrite each other. If the ranges written to and the versioning history were available, an improved merging algorithm could detect this situation and simply overwrite the other transaction instead of walking the entries of the tree to merge differences.

However, more sophisticated approaches would require to share the versioning history and operation logs, complicating things in distributed environments, where the algorithms presented in this thesis only require sharing the extents and their meta-data.

Extent Allocation

The algorithm for extent allocation can have a big impact on performance. The consequence in this thesis has been to use private allocation mode, where every process has an independent pool of extents to allocate from. A possible solution for shared extent pools could be to use techniques inspired by hash tables such as quadratic probing. A diverging or even randomized access pattern could significantly improve performance of the allocator.

6.1.2 POSIX-compliant File System

Implementing one of the possibilities described in section 4.5 allows for the construction of a POSIX compliant file system. For keeping meta-data, intermediary extents containing attributes as well as the root of the tree containing the data payload of the file can be inserted instead of directly pointing to the payload.

In terms of access semantics, each operation would have to be wrapped in its own transaction. For read access, the commit will be a no-op since no writes will be in the same transaction. It is important to start a new transaction for every operation, since POSIX requires a change to be visible for all subsequent read operations.

6.1.3 Distribution and Usage on Barrelfish

Goby's write-once behaviour in extent access allows for simple replication. However, extra care has to be taken to invalidate extents that are garbage collected on any participating nodes that have a copy of this extent. This can be done asynchronously since an extent is only garbage collected if no references to it exist. As long as the de-allocation notification is received before the extent is re-allocated and referenced there is no risk of reading stale data.

In terms of Barrelfish, this can be achieved by having a central management service tracking extent allocation and reference counts. Individual domains can allocate from their own extent pool and only have to notify the management service asynchronously. If a domain encounters a referenced extent it has not in its own cache, a bulk-transfer can be performed directly with the domain owning the extent in question or any other domain having a copy. The management service is responsible for notifying domains upon extent de-allocation. Since every participating domain could be holding a copy of the extent, this message has to be broadcast.

Bibliography

[1] fusion-io; flash-based storage solutions. `http://www.fusionio.com`.

[2] IEEE standard for information technology- portable operating system interface (POSIX) base specifications, issue 7. *IEEE Std 1003.1-2008 (Revision of IEEE Std 1003.1-2004)*, pages c1 –3826, 1 2008.

[3] A. Baumann, P. Barham, P.E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 29–44, 2009.

[4] J. Bonwick and B. Moore. Zfs: The last word in file systems. *online]/retrieved on Jan. 22, 2008] Retrieved from the Internet*, 2007.

[5] B. Callaghan, B. Pawlowski, and P. Staubach. NFS version 3 protocol specification. Technical report, RFC 1813, Network Working Group, 1995.

[6] RF Freitas and WW Wilcke. Storage-class memory: The next storage system technology. *IBM Journal of Research and Development*, 52(4.5):439–447, 2008.

[7] D. Hitz, J. Lau, and M. Malcolm. File system design for an NFS file server appliance. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 235–246. San Francisco, CA, 1994.

[8] Y. Ho, G.M. Huang, and P. Li. Nonvolatile memristor memory: device characteristics and design implications. In *Computer-Aided Design-Digest of Technical Papers, 2009. ICCAD 2009. IEEE/ACM International Conference on*, pages 485–490. IEEE, 2009.

[9] J. Katcher. Postmark: A new file system benchmark. Technical report, Technical Report TR3022, Network Appliance, 1997. www. netapp. com/tech_library/3022. html, 1997.

[10] Paul E. McKenney, Dipankar Sarma, Andrea Arcangeli, Andi Kleen, Orran Krieger, and Rusty Russell. Read-copy update. In *Ottawa Linux Symposium*, pages 338–367, June 2002. Available: `http://www.linux.org.uk/~ajh/ols2002_proceedings.pdf.gz` [Viewed June 23, 2004].

[11] Daniel Phillips. A directory index for ext2. In *Proceedings of the 5th annual Linux Showcase & Conference - Volume 5*, ALS '01, pages 20–20, Berkeley, CA, USA, 2001. USENIX Association.

[12] S. Raoux, GW Burr, MJ Breitwisch, CT Rettner, Y.C. Chen, RM Shelby, M. Salinga, D. Krebs, S.H. Chen, H.L. Lung, et al. Phase-change random access memory: A scalable technology. *IBM Journal of Research and Development*, 52(4.5):465–479, 2008.

[13] P.A.B.C.W. Reid, M. Wu, and X. Yuan. Optimistic concurrency control by melding trees. *Proceedings of the VLDB Endowment*, 4(11), 2011.

[14] O. Rodeh. B-trees, shadowing, and clones. *ACM Transactions on Storage (TOS)*, 3(4):2, 2008.

[15] S. Venkataraman, N. Tolia, P. Ranganathan, and R.H. Campbell. Consistent and durable data structures for non-volatile byte-addressable memory. In *Proc. of the 9th Usenix Conference on File and Storage Technologies (FAST)*, pages 61–76, 2011.