Mergeable persistent data structures

Benjamin Farinier, Thomas Gazagnaire, Anil Madhavapeddy

Mergeable persistent data structures

Benjamin Farinier[1], Thomas Gazagnaire[2] and Anil Madhavapeddy[2]

1: ENS Lyon
benjamin.farinier@ens-lyon.fr

2: University of Cambridge
thomas.gazagnaire@cl.cam.ac.uk
anil.madhavapeddy@cl.cam.ac.uk

Abstract

Irmin is an OCaml library to design purely functional data structures that can be persisted on disk and be merged and synchronized efficiently. In this paper, we focus on the *merge* aspect of the library and present two data structures built on top of Irmin: (*i*) queues and (*ii*) ropes that extend the corresponding purely functional data structures with a 3-way merge operation. We provide early theoretical and practical complexity results for these new data structures. Irmin is available as open-source code as part of the MirageOS project.

1. Introduction

Distributed version-control systems (DVCSs) are widely used to manage the source code of software projects. These systems are at the core of any workflows to produce and maintain large software stacks that involve many branches of development and maturity. DVCSs are used to keep track of change provenance and help to relate newly discovered bugs in the software to the context of the modification of the source-code which caused it.

We wish to see the same kind of workflow applied to data as well as source code. What if you could version-control a (mutable) persistent data structure, inspect its history, clone a remote state and revert it to a previous state? We find that using the existing tools around source-code DVCS and applying them to application data results in significant benefits to developing distributed applications. Instead of structuring our system like a conventional database, we instead provide the functionality as a library that can be linked and customised directly by the application logic.

We will first explain the design ideas behind Irmin [1] – a portable library written in OCaml – and describe the implementation of two interesting data structures that we built using the library. Irmin uses the same concepts as Git [2]: *clone*, *push*, *pull*, *branch*, *rebase*, ... and exposes them as a library that can be used by any OCaml application. It also features a bidirectional mapping with the Git format: any changes committed by an application are reflected in the associated Git repository, and any manual change in the Git repository is reflected back in the application via Irmin. This let users use the usual Git tools (`tig`, `gitk`, ...) to inspect and manipulate the data structure. We demonstrate this increased flexibility power comes with a very reasonable cost, thanks to the use of purely-functional data structures and (implicit) hash-consing.

Our work is a generalization of the Concurrent Revisions framework [1] to handle arbitrary shape of history graphs. It is also an extension of Conflict-free Replicated Data-types [5], with much simpler

[1] https://github.com/mirage/irmin
[2] http://git-scm.com/

data structure state: we use 3-way merge instead of encoding the sequences of operations in the state itself. White it has been shown that is possible to make any data structure persistent [2], we do not believe such a technique exists for *mergeable* data structures. Irmin therefore provides an immutable key-value store abstraction over which users of the library can develop their own mergeable data structures.

One of the main issues with version control that will be familar to users of Git is the handling of conflicts. It is the nightmare of any (sane) developer to see the familiar:

```
$ git merge master
Auto-merging lib/hello.html
CONFLICT (content): Merge conflict in lib/hello.html
Automatic merge failed; fix conflicts and then commit the result.
```

In Irmin, conflicts can appear in two different situations: when two nearby users are modifying the same value at the same time, and when a value has been changed in two remote locations with the propagation of changes resulting in a conflict. Irmin allows the application developer to deal with these situations using several techniques: (*i*) Conflict-free replicated data-types, (*ii*) Supplying a custom merge operator with the data structure and (*iii*) Explicit handling of conflicts as first-class citizen in the API.

2. Irmin Architecture

Irmin provides a mutable high-level interface built upon two backend stores.

The first backend store is the *block store*: this is a low-level key/value append-only store, where values are a sequence of bytes and keys are deterministically computed from the values (for instance using SHA algorithms). This mean that:

- if a value is modified, a new key/value pair is created: the resulting data-store is *immutable*
- if two data-stores share the same values, they will have the same keys: the store is *consistent*, the overall structure only depend on the stored data

The block store contains serialized values from application contents, but also structured data, like prefix-tree nodes and history meta-data. As the store is append-only, there is no remove function. The store is expected to grow forever, but garbage-collection and compression techniques can be used to manage its growth. This is not an issue as commodity storage steadily becomes more and more inexpensive.

The second backend store is the *tag store*: this is the only mutable part of the system. It is a key/value store, where keys are names created by users and values are keys from the block store, and can be seen as a set of named pointers to keys in the block store. This store is expected to be local to each replica and very small. The tag store is central for higher-level algorithms such as synchronisation and garbage collection.

The high-level interface is generated over the block store, the tag store and the application-specific data structures. It lifts immutable operations on the block store into a mutable prefix-tree, whose signature is given in Figure 1. The prefix tree `path` is usually a list of strings and node values are the user-defined mergeable `contents`.

Irmin allows the application developer to deal with conflicts using several tools. One of them is the use of data structures with custom merge operators. The idea is to give an abstraction of the Irmin low-level store as a high-level data structure. In addition of their classical operations, these data structure include the merge operation mentioned earlier. In this paper, we will considered queue and

```
module type S = sig
  type t
  type path
  type contents

  val read: t → path → contents
  (** Read the content at [path] in [t]. *)
  val update: t → path → contents → unit
  (** Replace the contents at [path] in [t] by [contents] if [path] is already
      defined and create it otherwise. *)
  val remove: t → path → unit
  (** Remove the given [path] in [t]. *)
  ...
end
```

Figure 1: High-level prefix-tree interface of Irmin, generated over the block store, the tag store and the application contents description.

rope data structures. These data structures and their associated operations have already been widely studied, even in a pure functional context [4]. *We do try not to compete with existing implementations of such data structures, but to extend them with an efficient and consistent merge operation.* The signature of the 3-way merge functions are defined in Figure 2.

```
type t = ...
(** User-defined contents. *)
type result = [ `Ok of t | `Conflict of string ]

val merge: old:t → t → t → result
(** 3-way merge functions. *)
```

Figure 2: IrminMerge signature. In queues and ropes signature, `IrminMerge.t` refers to the type of the above `merge` function.

As there are several ways to implement a merge function over these data structures, we first have to define what is a good merge operation:

> Given two data structures to be merged, the result of every operation applied on these structures since their common ancestor has to be present in the result of the merge operation.

Even though no formal proofs are provided, we believe that the merge algorithms detailed in this paper are consistent to this principle. We are planning to translate these algorithms into a proof assistant such as Coq in order to assert them more strongly.

3. Mergeable Queues

The first data structure we chose to implement is the queue, for two reasons. The first one is the wide use of this data type, and hence the large choice of possible usecases. A basic one could be a queue of operations to be executed in parallel and distributed between several operators. In case of overload, a new operator could spawn and clone the queue in order to share and ease the work of other operators.

The second reason is the simplicity of the merge behaviour representation. The merge principle can be – in the case of queues – specialized as follows:

- Every element popped in one of the two queues to be merged has to be absent in the resulting queue.
- All of the elements pushed in one of the queues have to be present in the result in the same order.

3.1. Implementation Overview

Operation	Read	Write	
Push	0	2	$O(1)$
Pop	2 on average	1 on average	$O(1)$
Merge	n worst case	1	$O(n)$

Figure 3: Cost of queue operations where n denotes the length of the queue. Read and write are expressed in number of memory access.

There are several efficient implementations of queues that can perform enqueuing (or *push*) and dequeuing (or *pop*) operations in $O(1)$ time. However, Irmin is built on an append-only low-level store. Such representations of the memory matches well with the functional programming model where the memory is immutable. For this reason we design these mergeable queues as functional queues. The exposed signature is presented in Figure 4.

```
module type IrminQueue.S = sig
  type t
  type elt

  val create : unit → t
  val length : t → int
  val is_empty : t → bool

  val push : t → elt → t
  val pop : t → (elt * t)
  val peek : t → (elt * t)

  val merge : IrminMerge.t
end
```

Figure 4: Queues signature. The `peek` function returns a `(elt * t)` because a normalization can occur during its execution. In order to avoid useless computations, we return a potentially updated queue.

A functional queue is composed of two simple linked lists. The first one contains elements that have been pushed onto the queue, and the second one those that will be popped. When the pop list is empty, the push list is flushed into the pop one. This operation is called *normalization*. Event though the normalization is a linear operation, each element of the queue has to be normalized only once. That is why – as it is reminded in Figure 3 – the amortized cost of operations on the queue is $O(1)$.

The implementation of mergeable queues is based on an Irmin store containing three types of element: `Index`, `Node` and `Elt`. Their type declarations are given in Figure 5.

```
type index = {
  push: int;
  pop: int;
  top: K.t;
  bottom: K.t;
}

type node = {
  next: K.t option;
  previous: K.t option;
  elt: K.t option;
  branch: index option;
}

type elt = Index of index | Node of node | Elt of V.t
```

Figure 5: Type declaration of mergeable queue structuring elements. The Irmin store is specialized in order to containing such elements.

`Index` are queue accessors. They are defined by four fields, `push`, `pop`, `top` and `bottom`. The two first fields, `push` and `pop`, are the number of pushes and pops applied to the queue since its creation. They are useful for the merge operation. The two others, `top` and `bottom`, are keys of the top and bottom element of the queue. `Node` are elements manipulated by queue operations. They are composed of four optional elements, `next`, `previous`, `elt` and `branch`. `next` and `previous` are keys to a potential preceding or following element.In practice, only one of these two can be not empty. `elt` is also an optional key which points to a value of the queue. The last field `branch` is an optional index, used only by the merge operation. Finally, `Elt` contains elements added to the queue.

The two first main operations on a queue are push and pop. The push operation adds a new `Elt` containing the pushed value, and a `Node` pointing to this element and the previous bottom element of the queue. It returns a new `Index` where the bottom element is the new created `Node`. The pop operation tries to read the top element of the queue. If the pop list is empty, the queue is normalized. Then it returns the value associated with the reading `Node` and an `Index`, where the top element is the following element of the reading `Node`. On average, there are two reads and three writes in the Irmin store for one push and one pop. Figure 6 shows an example of mergeable queues internals.

3.2. Merging

The other main operation is merging. The merging operation takes three arguments: two queues to be merged, `q1` and `q2`, and a common ancestor to those two queues called `old`. The resulting queue – called `new` – has to reflect the transformations from `old` to both `q1` and `q2`. This is done in three step.

First, elements of `old` which have been removed from `q2` are removed from `q1`. This is done without accessing these elements by using the `push` and `pop` values. In the same way, all elements of `old` which are still in `q2` are removed. Then, `q2` is concatenated at the end of `q1` by adding a new node where the field `branch` contains the index of `q2`. In the current implementation, the merge operation uses – in the worst case – a number of reads linear in the size of `old`, but always only one write. However, using binary random-access lists [4] instead of classical chained lists will reduce the overall number of reads to the logarithm of the size of the old.

For example, let's assume the following case:

```
old = {push = 6; pop = 2}, q1 = {push = 8; pop = 2}, q2 = {push = 7; pop = 4}
```

We can see that there are two elements of `old` which have been removed in `q2`, but which are still present in `q1`. So we have first to removed them. More other, we cannot directly concatenated `q2` at the end of `q1`. Indeed, there is still two remaining elements of `old` in `q2` which are already present in `q1`. Again, we have to removed them before the concatenation step. Just before this final step, the internal state is therefore in the following situation.

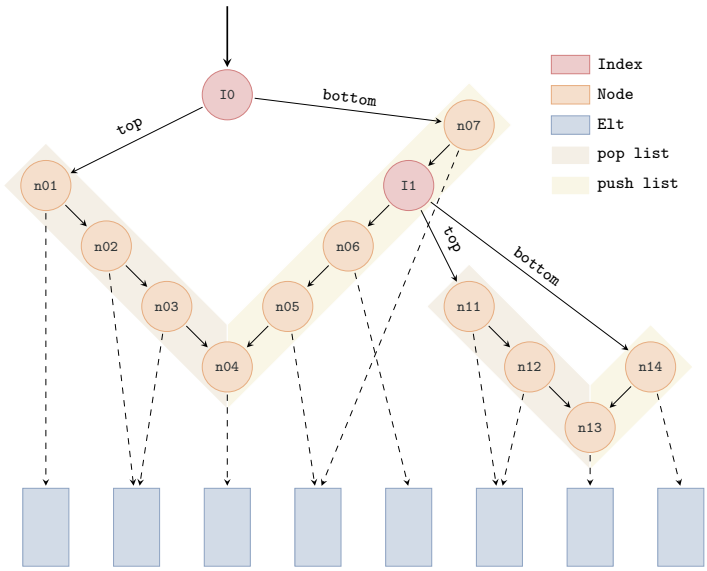

Figure 6: Example of a possible queue internal structure. Here, the main queue is accessible through the index `I0`. The index `I1` is pointing to a queue concatenated during a previous merge operation. This queue will be unfolded during the next normalization. Because of the Irmin store behavior, two nodes containing the same element share its physical representation.

```
q1' = {push = 8; pop = 4}, q2' = {push = 7; pop = 6}
```

The concatenation of those two intermediate queues return in the expected result. This step is done in constant time since we only need to add the index of `q2'` at the end of `q1'`.

4. Mergeable Ropes

A rope is a data structure that is used for efficiently storing and manipulating a very long string. A rope is a binary tree having leaf nodes that contain a short string. Each node has an index equal to the sum of the length of each string in its left subtree. Thus a node with two children divides the whole string into two parts: the left subtree stores the first part of the string and the right subtree stores the second part. The binary tree is crossed from the root to leaf each time the string has to be accessed. It can be done in $\log(n)$ time if the tree is balanced.

Operation	Rope	String
Set/Get	$O(\log n)$	$O(1)$
Split	$O(\log n)$	$O(1)$
Concatenate	$O(\log n)$	$O(n)$
Insert	$O(\log n)$	$O(n)$
Delete	$O(\log n)$	$O(n)$
Merge	$\log(f(n))$	$f(n)$

Figure 7: Comparison of the complexity of several operations on ropes and strings. n denotes the length of the rope/string, and f is the complexity of the merge function provided by the user.

The main operations on a rope are *set*, *get*, *split*, *concatenate*, *insert* and *delete*: Set and get

respectively set or get the character at a given position. Split(t, i) split at the position i the rope t into two new rope. Concatenate(t_1, t_2) return a new rope which is the concatenation of t_1 and t_2. Insert(t, i, s) insert the character chain s in the rope t at the position i. Finally, Delete(t, i, j) delete in the rope t the characters between i and j. Figure 7 compares the complexity of these operations for a rope and a string.

If ropes are mainly used to manipulate strings, they can also be used to manipulate any other type of container, as long as they support the above six operations. In fact, we design the ropes with the following idea: "give me a container with a set of operations, and we will return you a rope on this container, supporting the same operations but achieving a better complexity, with the exception of set and get". We therefore request a merge operation on the container in order to implement the merge operation of the mergeable rope. Indeed, because such a rope can be built on any type of container, it is impossible to have a general way to merge it.

This versatility was one of the major reason which motivated the choice of ropes as our second mergeable data structure implementation. Indeed, in a context such as MirageOS [3], the rope can be instantiated as a file system, using pages as rope atomic elements. In combination with Irmin, we obtain a Git-like persistent file system that can be distributed over several machines.

4.1. Implementation Overview

The implementation of mergeable ropes is quite straightforward, in the sense that it follows the previous description, and its signature is given in Figure 8. The tree containing the rope is a self-balancing binary search tree which keeps a factor of two between its minimal and maximal depth. To implement such tree, the Irmin store is specialized in order to contain three types of elements. As in mergeable queue, `Index` are accessors to the data structure. `Node` are intermediate elements of the tree which contain information to improve the binary search. Finally, `Leaf` are the user-defined container on which the rope is built.

```
module type IrminRope.S = sig
  type t
  type value (* e.g char *)
  type cont  (* e.g string *)

  val create : unit → t
  val make : cont → t
  ...
  val set : t → int → value → t
  val get : t → int → value
  val insert : t → int → cont → t
  val delete : t → int → int → t
  val append : t → t → t
  val split : t → int → (t * t)

  val merge : IrminMerge.t
end
```

Figure 8: Ropes signature. Here, `cont` and `value` can be instantiated as `string` and `char`, but also with any other possible `array` of `value` where basic function are written in a persistent style.

The implementations of the six main operations follow more or less the same pattern. The algorithm performs a binary search on the tree in order to determine the leaves concerned by the operation. Then the operation is applied on the containers found in the leaves. In order to achieve

the $\log n$ complexity, the size of these containers has to be of the same order as the depth of the tree. Finally, the tree is recursively rebuilt, using a rotation transformation in order to maintain the balancing property.

4.2. Merging

The merge operation is a bit different. Given two trees to be merged and their common ancestor, their keys in the Irmin store are used to determine the smallest subtrees where modifications occurred. This decision is recursively performed by the two functions detail in Figure 9 and Figure 10. Then, if these subtrees are separated, the resulting rope can be automatically deduced. Indeed, as they are separeted the modifications occurred in differentiable places, that means we can therefore include all of them being sure there is no conflicts. If it is not the case, these subtrees are linearised and flushed into containers on which the user-defined merge operation is applied.

```
let merge_branch old1 old2 (branch11, branch12) (branch21, branch22) =
  if (branch11.key = branch21.key) then Some branch11
  else if (branch12.key = branch22.key) then (
    if (old1.key = branch11.key) then Some branch21
    else if (old1.key = branch21.key) then Some branch11
    else None
  )
  else if (branch12.key = old2.key && branch21.key = old1.key) then Some branch11
  else if (branch11.key = old1.key && branch22.key = old2.key) then Some branch21
  else None
```

Figure 9: The key comparison function, which returns an option for `branch11` and `branch12` with the help of `branch21` and `branch22`. The top level conditional statement distinguishes three major cases. First, if `branch11` and `branch12` are equal, then we can return any of them. Second, if `branch21` and `branch22` are equal, then we can return a branch if the other one is still equal to `old`. Finally, if modifications occurred in different branches, we can return the modified branch.

```
let rec merge_node old node1 node2 =
    let left1 = node1.left in
    let right1 = node1.right in
    let left2 = node2.left in
    let right2 = node2.right in
  let left_option =
    merge_branch store old.left old.right (left1, right1) (left2, right2) in
  let right_option =
    merge_branch store old.right old.left (right1, left1) (right2, left2) in
  match (left_option, right_option) with
  | None, None → ... (* bad case *)
  | Some left, None | None, Some right → ... (* recursive case *)
  | Some left, Some right → ... (* good case *)
```

Figure 10: The merge decision function. The final pattern-matching distinguishes three case. The *good case* return immediately the result of the merge, since it achieve to deduce it at this step. The *recursive case* recall the merge function on the `None` subtree. The *bad case* appears when it is impossible to automatically deduce the result of the merge. In this case, the two subtrees are flushed into two containers, and the merge function provided by the user is called on it.

On simple trees, this approach is very efficient because the merge operation is applied on the

smallest possible container. However, the balancing property reduces this effectiveness. Indeed, the internal structure of a tree can be deeply modified during a rebalancing operation, making it impossible to compare potentially large subtrees. In order to minimize the scope of rebalancing operations, the rotation function is applied as less as possible, and only on the smallest unbalanced subtree. Due to this restriction, the impact on the merge function efficiency is proportional to the volume of modifications produced by an operation. An example of merge operations is given in Figure 11.

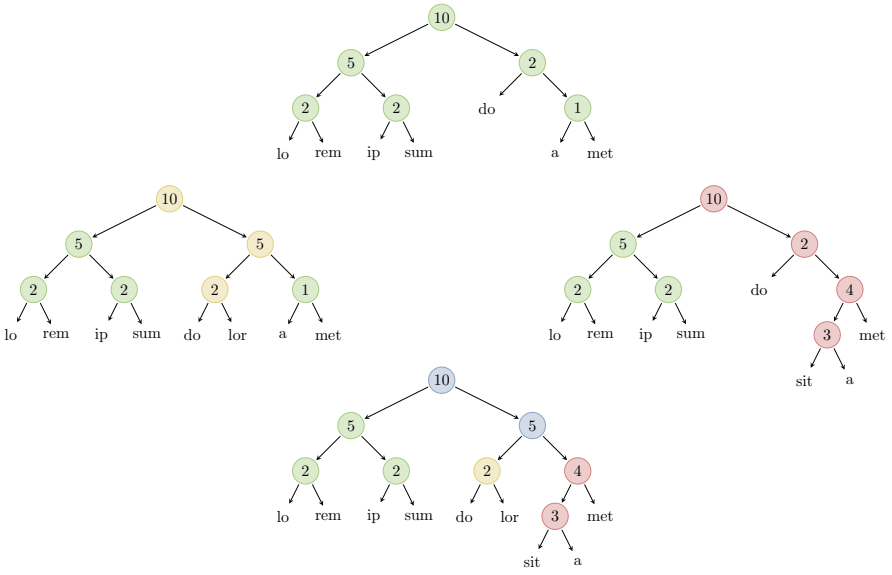

Figure 11: In this example, the old green rope have been modified in two different manners – yellow and red –, resulting in two ropes having to be merged. In this case, the merge algorithm is able to select the two relevant subtrees in both of the ropes, and returns the final rope without having to call the user-provided merge function.

5. Analysis

We now details how we evaluated the correctness of the previously described implementation and how we analyzed their effective costs.

5.1. Automatic checking

In the case of the correctness, we especially want to ensure that the classical operations match with their equivalents in other implementations, and that the merge operation follows its specification. The way it works is similar for queues and ropes. An oracle is used to determine a sequence of operations and their result. This sequence is then applied on the tested data structure, and on a witness obtained from another implementation. At each step, the data structure and the witness are required to exactly match. And at the end of the sequence, the two obtained result have to correspond with the result predicted by the oracle.

If this protocol is working fine with classical operations, it cannot be honestly applied on the merge operation. Indeed, it does not exist at our knowledge another data structure to be used as a witness. The verification process is therefore a bit different for this operation. In this case, we chose a result

with a easily verifiable property which is preserved after each merge operation. For example, a queue containing an increasing sequence of numbers, or a rope composed by a nondecreasing sequence of characters. This result is decomposed in several subresult having to be merge in order to recover the initial one. And because a merge preserves the aforementioned property, we can verify the correctness of the merge operation at each of its uses.

These tests have successfully guaranteed the good behaviors of our implementation of queues and ropes data structures. But they are not sufficient to ensure that the theoretical complexity is reached. In addition, we therefore designed several benchmark tests in order to validate this last point. Aside from showing the theoretical complexity is reached, these tests highlight some other interesting facts.

5.2. Benchmarking

Performance analysis The most obvious interest of benchmark tests is to measure the needed time of a given operation. Figure 12 shows the result of one of these tests. As a first glance, we can see that the theoretical complexity is reached for every operations. Then we can see a general stair behaviors. This one is due to the internal tree representation. Indeed, at each a step a new level is needed in this tree in order to represent the whole rope. Finally, we can see several spikes, which are the consequences of the complex interactions between the balancing property and maintaining a leaf length proportional to the depth of the tree.

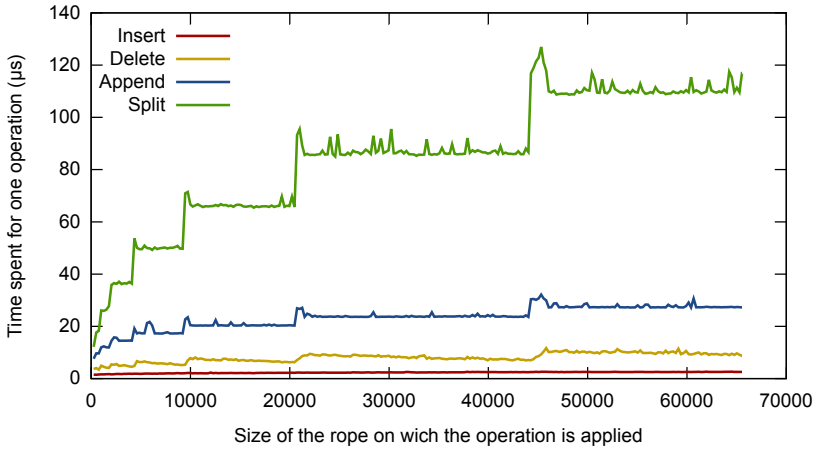

Figure 12: Needed time for one operation on a rope of size n on the Obj backend (see later)

Backend comparison As Irmin can be instantiated on different backends, it may be interesting to compare their relative performances. One of these backends is the OCaml heap, which maps Irmin blocks to the OCaml heap and use the memory address as block address (instead of its hash). This backend (whose implementation is given in Figure 13) uses the `Obj` module: it is useful to get raw performance results to compare it with already existing OCaml libraries. We do not advise to use it in other contexts.

Figure 14 shows the result of a same test run on different backends. On this graph, *Memory* refers to the in RAM backend. *GitMem* and *GitDsk* respectively refer to a Git backend, in the first case instantiate in the memory, on the hard drive in the second case. The *Core* is not strictly a backend because it refers in fact to the implementation of functional queue in the Core library. It is used here as a sort of optimal case, in a matter of comparison.

```
module ObjBackend ...  = struct
  type t = unit
  type key = K.t
  type value = V.t

  let create () = ()
  let clear () = ()

  let add t value =
    Obj.magic (Obj.repr value)

  let read t key =
    Obj.obj (Obj.magic key)

  let mem t key = true
  ...
end
```

Figure 13: ObjBackend implementation overview.

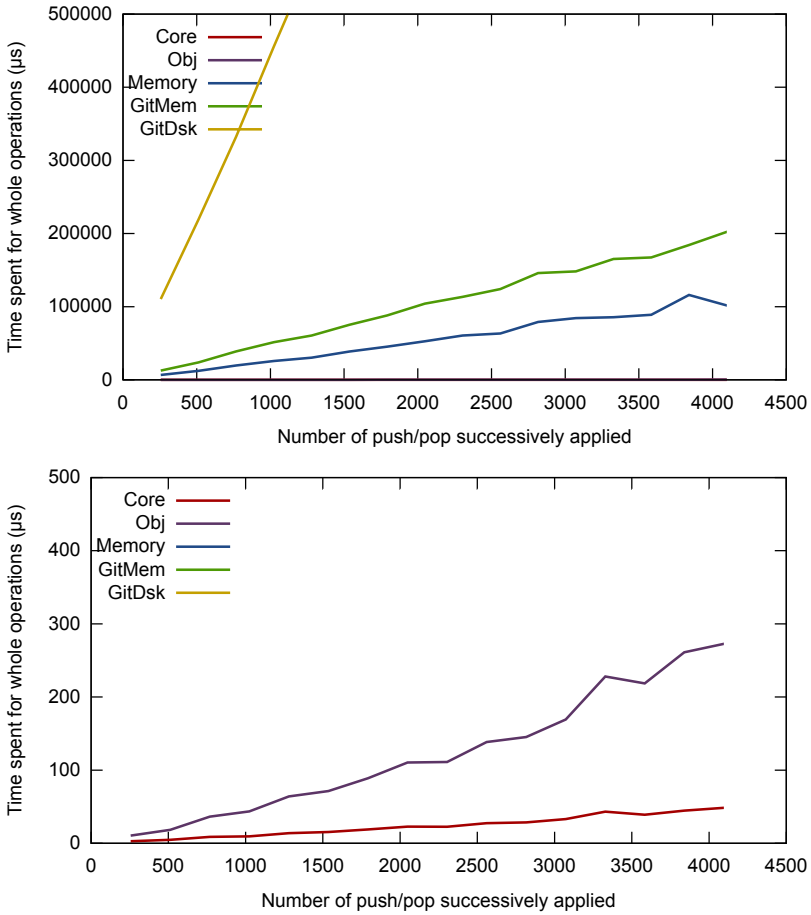

Figure 14: Time needed for n push followed by n pop on different backends

As we can see, there is a factor five between Core and the implementation of our queue on the Obj backend. It is an acceptable price to pay for maintaining a mergeable data structure.

IO cost estimation A final original use of benchmarks is the possibility to determine the time needed for a read and a write on different backends. Indeed, our rope implementation is able to produce some statistics about the number of read and write used in each operation. Some results of this statistics are given in Figure 15.

Aside from highlighting the obvious fact that the cost of an operation is proportional to the number of read and write, we can see that the relative proportion of read and write is different in each operation. Knowing the time needed for one operation, these proportions give us a linear system of four independent equation where variables are the time needed for a read and a write, represented in Figure 16. The average of intersection points indicates the solution that we looking for.

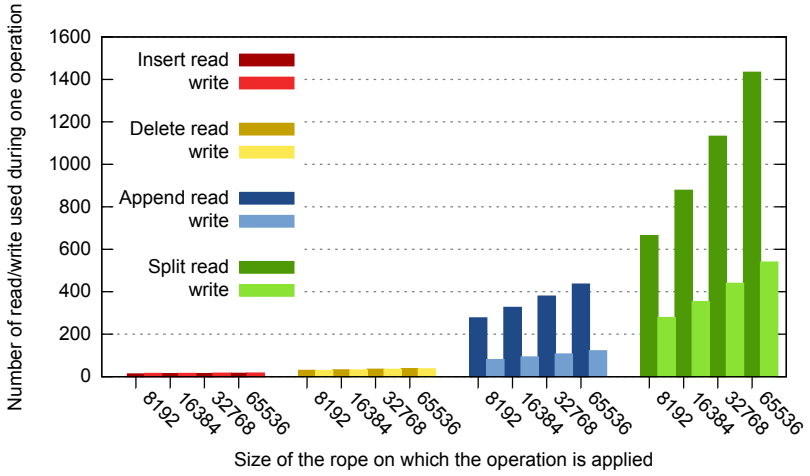

Figure 15: Number of read/write used during one operation on a rope of size n.

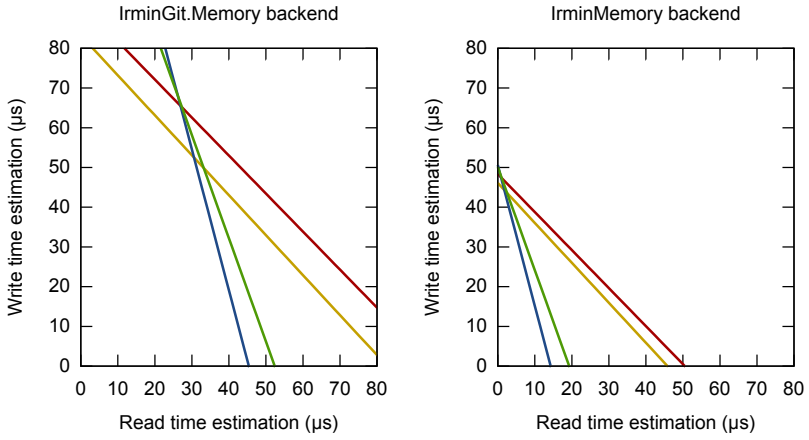

Figure 16: Let x be the time needed for one read, y the time needed for one write and d the time needed for one operation on a rope. Then the above curves – where colors match with the previous figure – are the plot of the equation: $x * \text{nrb of read} + y * \text{nbr of write} = d$

6. Conclusion

We have presented Irmin, a high-level OCaml library that follows the same design principles as Git. We presented the design, implementation and evaluation of two mergeable data structures: queues and ropes. We also evaluated the performance of the different backends of Irmin and showed that it is possible to build very efficient version-controlling data structures. As a future work, we plan to (*i*) build more mergeable data structures, (*ii*) build higher-level protocols on top of Irmin to manage a distributed computation and (*iii*) build eventually-consistent applications using the provided data structures and protocols. All of the source code is available under a BSD license at https://github.com/mirage/mergeable-queues and https://github.com/mirage/mergeable-ropes.

7. Acknowledgment

The research leading to these results has received funding from the European Union's Seventh Framework Programme FP7/2007-2013 under Trilogy 2 project, grant agreement number 317756.

References

[1] Sebastian Burckhardt and Daan Leijen. Semantics of concurrent revisions. In *Proceedings of the 20th European Conference on Programming Languages and Systems: Part of the Joint European Conferences on Theory and Practice of Software*, ESOP'11/ETAPS'11, pages 116–135, Berlin, Heidelberg, 2011. Springer-Verlag.

[2] James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making data structures persistent. *J. Comput. Syst. Sci.*, 38(1):86–124, February 1989.

[3] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library operating systems for the cloud. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 461–472, New York, NY, USA, 2013. ACM.

[4] Chris Okasaki. *Purely Functional Data Structures*. PhD thesis, Pittsburgh, PA, USA, 1996. AAI9813847.

[5] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, SSS'11, pages 386–400, Berlin, Heidelberg, 2011. Springer-Verlag.