POPL: U: Verification of a Cache-optimized Data Structure

Anonymous Author(s)

Abstract

Recent years have witnessed a rapid development of main-memory database systems thanks to the growingly affordable memory. DeepSpecDB is a main-memory database management system implemented in C with deep specifications and end-to-end verification guaranteeing the correctness of the system. In this paper, we will present current status of DeepSpecDB on verification of index data structures.

Keywords formal verification, database system

1 Introduction

As the unit price of memory decreases over time, efforts have been put into migrating the traditional disk-based applications into main memory. A classical category of such applications is databases. For example, H-store [6], MongoDB, Redis, and Memcached are well-known database systems featuring main-memory index and storage.

Meanwhile, database systems are the kind of critical applications where security vulnerabilities can lead to catastrophic consequences. We believe that formally verifying the functional correctness of programs with respect to deep specifications is the remedy. Deep specifications are rich, live, formal, and two-sided, ensuring that the behavior will be captured by the specifications and proved correctly by machine-checked proofs. With the help of the Verified Software Toolchain [1], we are now able to verify C programs against the CompCert [7] operational semantics. The DeepSpecDB project aims to design and verify a main-memory database system with deep specification guaranteeing the correctness of the system. In this paper, we present our efforts on the indexing data structures, which databases use to organize the data and speed up data access.

SRC Grand Finals '19, ,

2 Related Works

Many efforts have been put on formal verification of database system components. Véronique Benzaken and Évelyne Contejean have formalized semantics of relational databases [3] [4]. Verification of index data structures such as binary search trees is also common, such as works done by Tobias Nipkow [9].

This project distinguishes itself from other related works in that it bridges the gap between database systems and indexing data structures. We modify the interface of the data structures to support database operations, and we refine the specification all the way down to actual implementation in C.

3 Overview of the data structure

We design our database index based on MassTree [8]. In general, the data structure maps from variable-length strings to values. MassTree maintains the mapping by organizing strings in a *trie over B+ trees*: each node of the traditional trie is now replaced with a B+ tree and a trie node now corresponds to a fixed-length slice of key rather than a single character. To account for the possible residual suffix of keys, a data structure named *border node* is introduced. The border node replaces the leaf node of the original B+ tree, and can potentially point to the next layer of trie nodes or client values corresponding to different prefixes of current slice. The length of the slice depends on the CPU word size so that we can express the slice as an integer and speed up the operations. The fanout of the B+ trees and other parameters of MassTree are chosen so that internal structures can fit well in cache line which improves the performance. Although the original MassTree was designed for key-value databases, later development of Silo [11] and SiloR [12] suggest its capability of indexing for relational databases.

In the original MassTree design, the border nodes are linked with each other to facilitate range queries and removals. We decide to remove the links in favor of a new structure called *cursor* inspired by SQLite. Abstractly, if all keys are organized in an ordered list, a cursor should point between two adjacent keys and any key bounded by the two can be associated with the cursor. Concretely, the cursor for B+ trees is implemented with a list of pointers pointing at various internal nodes and the border node, and the indexes into those nodes. The list should also be the trace when one tries to access the border node given any slice of key associated with the cursor from the root node. The cursor for a trie is a list of B+ tree cursors, which is also the trace when one tries to access the client value given any key associated with the cursor.

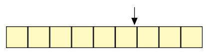

Figure 1. An abstract cursor

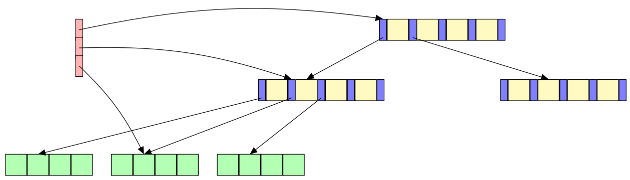

Figure 2. A B+ tree with a cursor

Cursors provide similar sequential performance of range queries to links because one can access the next or previous record in the structure by traversing upward and then downward with the help of cursors, which should cost only amortized constant time. Moreover, the traversal should hit cache frequently considering the locality of range queries. We favor cursors over links in the data structure because we anticipate better concurrent performance compared to implementation with links for two reasons. First, cursors can be allocated as thread local objects leading to fewer contentions. Second, we expect cursors to perform better when overwriting update is not available (updating the links, in this case, is expensive), which happens in the optimistic concurrency control variant from Silo.

4 Modular Coq-based verification using VST

The verification proceeds in a modular approach. In the database, the B+ trees are also used as the index for integer keys, and we build tries on exactly the same interface exposed to the rest of the database. We organize the verification in the same structure as we write the program. The verification of B+ trees [2] abstracts the implementation into specification. Tries are then verified given only the specification of B+ trees: the verification holds for any data structure that is verified with the same specification (shown in figre 3). Furthermore, the specification for the database index is parameterized by key types, which means the specification of tries is the same as that of B+ trees modulo the theory of keys. This also enables us to use the verification of trie as a test for the specification: only if the specification is neither over-restricted nor under-specified can the proofs go through.

In another direction, the application domain proof is modularized away from the concrete semantics in verification of both data structures. That is, C programs are proved to correctly implement corresponding functional programs. The

```
Module Type KV_MAP (KeyType: UsualOrderedType).
  Definition key := KeyType.t.
  Parameter map: Type -> Type.
  Parameter cursor: Type -> Type.

  Section Types.
    Context {value: Type}.
    Parameter empty: map value -> Prop.

    Parameter make_cursor: key -> map value -> cursor value.
    Parameter first_cursor: map value -> cursor value.
    Parameter last_cursor: map value -> cursor value.
    Parameter next_cursor: cursor value -> map value ->
    ↪  cursor value.
    Parameter prev_cursor: cursor value -> map value ->
    ↪  cursor value.

    Parameter get: cursor value -> map value -> option (key *
    ↪  value).

    Parameter put: key -> value -> map value -> (* input *)
                   map value -> (* output *)
                   Prop.
    [...]
  End Types.
End KV_MAP.
```

Figure 3. Interface

refinement is established in Verifiable C using Verified Software Toolchain. The functional programs are proved to conform to the specification, which is usually properties about keys, values, and interaction with the data structures. The benefit of modularization is significant: we need not care about the low-level details, such as pointers and memory management when reasoning about functional correctness. We can also forget the application logic when we are in the separation logic world.

Figure 4 demonstrates the modularity of proofs, with arrows indicating dependencies and the thick line denoting the boundary of index data structures (as compared to the entire database system).

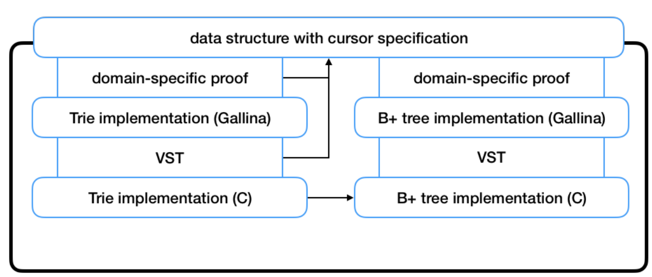

Figure 4. Modular Verification

5 Insights

5.1 Modeling Cursors with Keys

We assume that keys form a totally ordered set, and we view an indexing data structure abstractly as an ordered *kv-map*. In our model, operations on the kv-map are based on cursors. One can make a cursor from a key. One can obtain the first cursor or last cursor in the kv-map. One can move a cursor forward or backward in the kv-map. One may get the kv-pair from the kv-map located by the cursor. Informally, the semantics of the get operation is defined to return the first pair to the increasing direction of the cursor. In the special case where a cursor is made by a key also in the kv-map, it will return the exactly kv-pair containing the key in this definition, which resembles the usual semantics of a kv-map.

To verify data structures with cursors, we need to translate the informal intuition of cursors into formal definitions. A map containing n keys $(k_1, \ldots, k_n)$ divide the entire key space into $n + 1$ equivalent classes. Equivalent class K_i is defined by the range $(k_i, k_{i+1}]$, with the special case of $K_0 = (-\infty, k_0]$ and $K_n = (k_n, +\infty)$. We take k_i as the representative of class K_i. We then define a key and a cursor is associated if the key is a member of class K_i and get operation returns a kv-pair containing key k_i. Alternatively, we can define a cursor to be associated with class K_i instead.

In code, we define the equivalence class by quantification over key space, as the two properties in the figure 5.

```
Definition get_key (c: cursor) (m: map):
  match get c m with
  | Some (k, _) => Some k | None => None
  end.

Axiom get_ge: forall (k k': key) (m: map),
  get_key (make_cursor k m) m  = Some k' -> k' >= k.

Axiom get_least: forall (k1 k2 k1' k2': key) (m: map),
  get_key (make_cursor k1 m) m = Some k1' ->
  get_key (make_cursor k2 m) m = Some k2' ->
  k1 <= k2 <= k1' -> k1' = k2'.

Definition cursor_key_assoc (k: key) (c: cursor) (m: map):
  get_key c m = get_key (make_cursor k1 m).
```

Figure 5. Formalization of Equivalence Classes

We realize that there might be multiple cursors associated with an equivalent class of keys, but they should be indistinguishable. For example, if the kv-map (implemented in trie) contains keys of ("hello", "world"), cursors made by keys of "proof" or "query" might be structurally different from each other, but they should be equivalent with respect to the operations. This leads to the definition of equivalence relation of cursors: Any pair of cursors associated to the same key is defined to be equivalent. This definition is equivalent to the weaker form where associated with equivalent keys are acceptable.

We can now extend the definition to cover other operations. If a cursor is associated with class K_i, moving forward will return a cursor associated with class K_{i+1} and similar for backward. The first cursor is associated with class K_0 and the last is associated with class K_n.

5.2 Augmented types

When using separation logic, one needs to derive the abstract data types corresponding to the concrete ones, where all the addresses are typically abstracted away (for example, in the proof for list reversal [10]). However, a cursor might point to internal nodes of a data structure which is often achieved by saving copies of pointers in C. We find it hard to express the same relation using the abstract cursor and the abstract data structure. One promising solution is to use magic wand [5], but this approach turns out to be unsound. Therefore we introduce the augmented types, where we include the addresses in the abstract data type in order to express the relation between the cursor and the data structure without violating the soundness.

Augmented types, however, introduce another problem: the addresses of data structures are usually determined by allocation at runtime and data structures located in different address space can express the same abstract data. To account for the non-determinism, we type our specifications for mutating operations as predicates rather than functions, as shown in figure 6. It would be ideal if we can still justify the determinism excluding the addresses, which we will see in the next section.

```
Module Type KV_MAP (K: DecidableType).
  [...]
  Variable value: Type.
  Parameter get: cursor -> map -> option value.
  Parameter put: K.t -> value -> map -> map -> Prop.
  [...]
End Table.
```

Figure 6. Difference between deterministic and non-deterministic operations

5.3 Flattening

Coq has a very strict type system to ensure the strict positiveness of inductive types and termination of recursive functions. It ensures the soundness of the system, but also adds to the difficulty to define trie over B+ trees: with B+ trees hidden from trie's definition, Coq cannot check the strict positiveness and rejects the definition. We introduce the concept of *flattening* to pass the type check. In short, an extra function flatten is required in the specification which turns an abstract data structure into an ordered list of key-value pairs containing the same set of data. We can then define the data structure for trie based on flattened B+ trees, which can pass Coq's type check without problems.

A flattening function should maintain the property that it produces an ordered list of key-value pairs containing the same set of data. The orderedness can be formalized rather easily, but the "same set of data" is hard to encode using the current language: we can only define it through operations with cursors. However, we shall see that the original map and the flattened map contains exactly the same set of data if and only if their key space is divided into the same equivalence classes. Therefore we define the relation by stating that for any key and any cursors for corresponding maps associated with this key, they should equivalent with respect to the get operation. If the property is violated, we know that the boundary of equivalence classes are not aligned and therefore data is different. The formal code is presented in figure 7.

```
Theorem flatten_invariant: forall (m: map),
  ordered m /\
  forall (k: key) (c1 c2: cursor),
  cursor_key_assoc k c1 m ->
  cursor_key_assoc k c2 (flatten m) ->
  get c1 m = get c2 (flatten m).
```

Figure 7. Formalization of flatten invariant

Flattening excels among other candidate solutions to the above problem because it synergizes with other components of verification well. We find similar ideas in Tobias Nipkow's verification of search trees [9], where flattening is used to verify the functional correctness. We find it also helpful in our verification: many theorems would have been proved separately for different data structures can now be proved once for the flattened list and work for all. For example, the theorem `Forall_put` and `Forall_get` in figure 8 and their counterparts in separation logic are useful for recursive types and client data structures to the map. Flattening also implies the determinism of data structure mutations: if the flatten invariant is preserved, it is already proved that the flatten operation can commute with the put operation. While the put operation for the original map might be non-deterministic, put operation for the flattened map is excluded without any non-determinism.

6 Conclusions

In this project, we achieve verification of a high-performance kv-map. We establish the end-to-end verification between the low-level implementation in C and the high-level specification. We demonstrate that formal verification is feasible for large-scale and complicated code base using modular approach. Furthermore, this project suggests that verification is achievable without any sacrifice of performance.

```
(* the deterministic version of put for flattened map,
↪  defined by the library *)
Definition flattened_put (k: key) (v: value) (m:
↪  flattened_map) := [...].
Theorem put_commute: forall (m1 m2: map) (k: key) (v: value),
  put k v m1 m2 ->
  flatten m2 = flattened_put k v (flatten m1).
Theorem Forall_put (P: key * value -> Prop):
  forall (m1 m2: map) (k: key) (v: value),
    P (k, v) -> Forall P (flatten m) -> put k v m1 m2 ->
    Forall P (flatten m2).
Theorem Forall_get (P: key * value -> Prop):
  forall (m: map) (c: cursor) (k: key) (v: value),
    Forall P (flatten m) -> get c m = Some (k, v) ->
    P (k, v).
```

Figure 8. Consequences of Flatten Invariant

References

[1] Andrew W Appel. 2011. Verified software toolchain. In *European Symposium on Programming*. Springer, 1–17.

[2] Aurèle Barrière. 2018. VST Verification of B+Trees with Cursors. (2018).

[3] Véronique Benzaken and Évelyne Contejean. 2019. A Coq Mechanised Formal Semantics for Realistic SQL Queries: Formally Reconciling SQL and Bag Relational Algebra. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2019)*. ACM, New York, NY, USA, 249–261. https://doi.org/10.1145/3293880.3294107

[4] Véronique Benzaken, Evelyne Contejean, Ch. Keller, and E. Martins. 2018. A Coq Formalisation of SQL's Execution Engines. In *Interactive Theorem Proving - 9th International Conference, ITP 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings (Lecture Notes in Computer Science)*, Jeremy Avigad and Assia Mahboubi (Eds.), Vol. 10895. Springer, 88–107. https://doi.org/10.1007/978-3-319-94821-8_6

[5] Qinxiang Cao, Shengyi Wang, Aquinas Hobor, and Andrew W Appel. 2018. Proof Pearl: Magic Wand as Frame. (2018).

[6] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan PC Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, et al. 2008. H-store: a high-performance, distributed main memory transaction processing system. *Proceedings of the VLDB Endowment* 1, 2 (2008), 1496–1499.

[7] Xavier Leroy. 2006. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *ACM SIGPLAN Notices*, Vol. 41. ACM, 42–54.

[8] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM european conference on Computer Systems*. ACM, 183–196.

[9] Tobias Nipkow. 2016. Automatic functional correctness proofs for functional search trees. In *International Conference on Interactive Theorem Proving*. Springer, 307–322.

[10] John C Reynolds. 2002. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on*. IEEE, 55–74.

[11] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 18–32.

[12] Wenting Zheng, Stephen Tu, Eddie Kohler, and Barbara Liskov. 2014. Fast Databases with Fast Durability and Recovery Through Multicore Parallelism.. In *OSDI*, Vol. 14. 465–477.