Smarter Contract Upgrades with Orthogonal Persistence

Luc Bläser
luc.blaeser@dfinity.org
DFINITY Foundation
Switzerland

Claudio Russo
claudio.russo@dfinity.org
DFINITY Foundation
Switzerland

Gabor Greif
gabor.greif@dfinity.org
DFINITY Foundation
Switzerland

Ryan Vandersmith
ryan.vandersmith@dfinity.org
DFINITY Foundation
Switzerland

Jason Ibrahim
jason.ibrahim@dfinity.org
DFINITY Foundation
Switzerland

Abstract

Altering the smart contract deployed on a blockchain is typically a cumbersome task, necessitating a proxy design, secondary data storage, or the use of special APIs. This can be substantially simplified if the programming language features orthogonal persistence, automatically retaining the native program state across program version upgrades. For this purpose, a customized compiler and runtime system needs to arrange the data in a self-descriptive portable format, such that new program versions can pick up the previous program state, check their compatibility, and support implicit or explicit data evolutions. We have implemented such advanced persistence support for the Motoko programming language on the Internet Computer blockchain. This not only enables simple and safe persistence, but also significantly reduces the cost of upgrades and data accesses.

CCS Concepts: • **Software and its engineering** → **Runtime environments**; • **Information systems** → **Main memory engines**.

Keywords: Orthogonal Persistence; Smart Contract Upgrades; Blockchain; WebAssembly

ACM Reference Format:
Luc Bläser, Claudio Russo, Gabor Greif, Ryan Vandersmith, and Jason Ibrahim. 2024. Smarter Contract Upgrades with Orthogonal Persistence. In *Proceedings of the 16th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (VMIL '24), October 20, 2024, Pasadena, CA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3689490.3690401

1 Introduction

Modern blockchains, like the Internet Computer [2, 11], establish a secure and distributed virtual machine for running complex programs, such as smart contracts, decentralized applications, or other software solutions. Such blockchain programs can be implemented in a Turing-complete high-level programming language, such as for example the blockchain-tailored languages Solidity [10] and Motoko [18], or mainstream languages like Rust, JavaScript, Python, and others.

Even when deployed on a blockchain, there typically comes a time when a program needs to be changed, be it for feature extensions, improvements, or bug fixes. This requires a mechanism to upgrade a program's code, while retaining its state, replacing an existing version by a new version that implements the desired change. Unfortunately, such support is typically lacking or poor, requiring programmers to apply "creative" alternative solutions or implementing cumbersome storage management. On blockchains, like Ethereum [10], programmers usually prepare proxies to enable upgrades by changing the redirection target [17]. More advanced blockchains support a dedicated upgrade mechanism [13]. However, the integration in the programming language is still influenced by the traditional computer architecture, where the program has a main memory that is lost on an upgrade. As a consequence, the blockchain often exposes extra secondary memory where any more long-term data can be stored to survive program upgrades. This not only complicates program logic, but also incurs safety and performance disadvantages. We discuss these techniques in more detail in Section 2.

In this work, we strive to radically simplify program upgrades on a blockchain, by achieving high flexibility, performance, and safety at the same time. The idea is that the program state, implemented in the standard inbuilt concepts of the language, such as object-orientation, is automatically persisted on the blockchain, and is even retained across program version changes. This property is known as *orthogonal persistence* [3–8, 15, 16]: The program lives conceptually indefinitely, its state is implicitly persisted without the explicit use of a database or secondary storage. The programming language provides the necessary support to evolve both the program logic and the persisted data representation.

Of course, this cannot be achieved by a classical programming language implementation but requires a dedicated compiler and runtime system, that, for example, avoids any static allocation and places all data in a dynamic heap with sufficient metadata. On a program upgrade, the new version can then pick up the existing memory, check compatibility, perform or guide necessary data migration, and then continue program execution with the new version.

We have implemented such orthogonal persistence for Motoko [18], an expressive high-level programming language that runs on top of the Internet Computer [2, 11]. It realizes a self-descriptive heap persisted on the blockchain in a format that allows subsequent program versions to check compatibility and resume operation if safe. The runtime system supports automatic data migration for well-defined, common change patterns, and beyond that, allows users to explicitly program any kind of more complex migration.

Compared to other programming languages for the IC, Motoko's orthogonal persistence offers these benefits:

- **Simplicity**: Developers no longer need to write specific logic for cross-upgrade persistence; a single keyword suffices to declare data for retention.
- **Safety**: The runtime system checks data compatibility on an upgrade, with the possibility of explicit migration if necessary. There is no way to misinterpret or corrupt the data persisted by a previous version.
- **Performance**: The upgrade does not modify the heap but only checks compatibility based on a type table. As a result, the upgrade is very fast, only depending on the number of static types, and not on the number of objects.
- **Flexibility**: Programmers are at liberty to select almost any types for their persistent data and are not restricted to a small selection of persistent representations (e.g. a key-value store). Currently, the only restriction is that the persisted types must not contain implicit references to code (excluding e.g. function types). Migration to any other structure is possible, either implicitly, when allowed by subtyping or explicitly, by user-defined coercion.

In summary, we make the following contributions:

- The design of a compiler and runtime system with efficient and safe orthogonal persistence running on a blockchain.
- The open-source implementation of orthogonal persistence for the programming language Motoko, running on the Internet Computer.
- An experimental evaluation of the orthogonal persistence in comparison to traditional upgrade techniques on the Internet Computer.

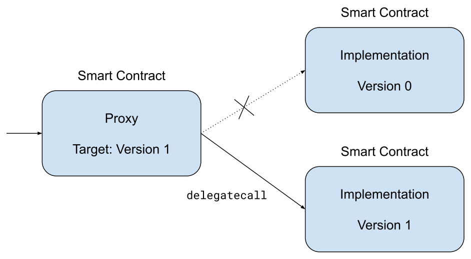


Figure 1. Enabling Ethereum contract upgrades via a proxy

The remainder of this paper is organized as follows: Section 2 provides background information about prevalent upgrade techniques used on blockchains, as well as our implementation target. Section 3 describes how our programming language integrates orthogonal persistence. Section 4 explains the design and implementation of the compiler and runtime system to support the persistence. Section 5 reports on experimental results of orthogonal persistence in comparison to classical persistence techniques. Section 6 discusses related work. Section 7 finally concludes this paper.

2 Background

To provide sufficient background information for the subsequent sections, we first discuss how program upgrades are currently enabled on blockchains. We then provide more details on our implementation platform.

2.1 Upgrade Techniques

Like in traditional software development, on a blockchain, one occasionally needs to alter a program's code. This is enabled by an *upgrade*, replacing the previous program logic by a new version while keeping the relevant state.

2.1.1 Call Redirection. Some blockchains like Ethereum [10], forbid the alteration of a deployed smart contract. Instead, programmers need to prepare upgrades by a mechanism of call redirection. A common pattern is to put a proxy program in front of the actual contract that contains the actual implementation [26], see Figure 1. Users are then instructed to always refer to the proxy which eventually redirects their calls to the actual smart contract. A program update can now be realized by resetting the redirection address in the proxy such that it points to the new smart contract, implementing the new version. By using the Ethereum virtual machine instruction `delegatecall`, the proxy can execute the call in the target contract by impersonating the caller. Data is usually stored in the proxy, imposing restrictions on the data evolution paths, e.g. changing the existing persistent data structure is not possible.

Apparently, the proxy patterns and other similar redirection mechanisms do not only add development complexity but also pose a security risk: For example, a program upgrade only depends on a value stored in the proxy (or other routing

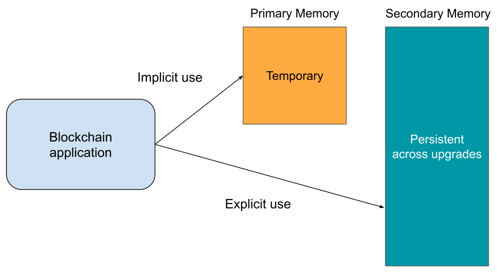


Figure 2. A blockchain program operating on two memories

logic) that may be replaced without sufficient transparency and control of all program users. Moreover, various design aspects need to be considered to avoid breaking interface changes and to retain flexibility for future changes [17].

2.1.2 Secondary Memory. Other blockchains [2, 28] offer a dedicated mechanism for upgrading a program that is already deployed on the blockchain. The upgrade is an explicit operation that follows defined authorization and if desired, governance rules, e.g. that the user community can review the change and vote on an upgrade to be either accepted or rejected.

However, while the upgrade process is supported at the blockchain level, its technical integration in the programming language remains quite limited. When the program uses the standard language features for managing state, such as object-orientation, this state is lost on an upgrade. Instead, programmers need to explicitly preserve the state across upgrades. This is typically done by storing the data to a secondary memory, using specialized data structures, or manually copying state on upgrades by special API hooks. The underlying design is reminiscent of traditional computer architecture. The blockchain usually exposes two types of memories, see also Figure 2:

- **Primary memory**: The main memory for the actual program state used by the program execution as its language-native memory. In some blockchains, this is not stored in the blockchain and discarded at the end of a transaction [28]. In other blockchains, like the IC, the state is persisted on blockchain but it is nevertheless erased on an upgrade because classical languages cannot interpret the memory image of a previously compiled program version.
- **Secondary memory**: A secondary (also called stable or global memory) that is explicitly managed by the program or through some dedicated data structure API. This memory is certainly persisted on the blockchain and can be consulted after an upgrade. Inspired by traditional computers, this memory can also be larger than the main memory, even if both memories are equally stored on the blockchain.

Using secondary memory for persistence has several downsides:

- Extra code is required to store the state. If omitted, the data is only transient and lost on an upgrade.
- The data needs to suit the available data structures for secondary memory, e.g. a key-value store.
- The data needs to be expensively serialized and deserialized to and from secondary storage.
- There is the risk of corrupting or misinterpreting the secondary storage, e.g. when changing the data types with a new program version.
- The secondary memory needs to be managed.

2.2 The Internet Computer

The Internet Computer (IC) [2, 11] is a powerful blockchain that allows running large-scaled decentralized software. Programs are organized as software components, called canisters, that realize the actor model [22] and use WebAssembly (Wasm) [20] at the execution level. Each canister has substantial compute and storage resources. The IC supports around 2 billion instructions per second per canister and offers memory of around 400 gigabytes per canister. Operations are issued as transactions that have a hard instruction limit and change the canister state atomically on success, or roll back on failure.

Various programming languages can be used to implement canisters, such as mainstream languages like Rust and TypeScript, among others, or Motoko, a specialized language for the IC (explained in Section 2.3).

For historical reasons, and to support classical languages without dedicated persistence across upgrades, the IC also exposes two memories per canister:

- **Main memory**: The primary memory that preserves the native program state between calls to the same canister. Although this state is stored on the blockchain, the state used to be discarded on an upgrade, because none of the available languages could exploit it for persistence. Moreover, until recently, the main memory was limited to 4 GB because of an underlying 32-bit address space imposed by the initial release of Wasm.
- **Stable memory**: A secondary memory that is also persisted on the blockchain and specifically serves to save the state that needs to survive the upgrade. The stable memory can be accessed by a low-level API [14] or through a stable data structure library [19]. It supports 400 GB in a 64-bit address space.

We recently adjusted this memory architecture to leverage main memory for our envisioned orthogonal persistence support, see Section 4.1.

2.3 Motoko

Motoko [18] is a relatively young programming language, tailored to the IC runtime model, to ease development. It

```
actor {
   stable var entries = List.nil<Text>();
   stable var times = List.map<Text, Time.Time>(entries, func e { 0 });
   public func log(t : Text) {
       entries := List.push(t, entries);
       times := List.push(Time.now(), times); // record times
   };
   public query func readLast(count : Nat) : async [Text] {
       List.toArray(List.take(entries, count));
   };
   public query func readUntil(t0 : Time.Time) : async [Text] {
       let after = List.filter<Time.Time>(times, func t { t >= t0 });
       List.toArray(List.take(entries, List.size(after)));
   };
};
```

Figure 3. A sample logging actor, original version and highlighted upgrade.

offers imperative, functional, object-oriented, and asynchronous programming concepts, while integrating IC-specific concepts directly in the language, such as the actor model and, being in the focus of this work, orthogonal persistence. Motoko emphasizes static typing and manages memory with a garbage collector [9]. It has a rich type system with the usual set of numeric types, tuples, options, variants, mutable and immutable records and arrays, recursive types, and polymorphic functions as well as actor-oriented futures, actor functions and actor types. The type system supports structural subtyping.

3 Programming Model

As a unique feature in the blockchain space, the Motoko programming language incorporates orthogonal persistence as a core concept: Programmers can simply use the standard concepts of the language to manage program state of an arbitrary structure, and this state is automatically persisted, even across upgrades. There is no need for secondary storage, special data structures, or other database-like abstractions.

To give a taste of the language, Figure 3 defines an actor that can be used to maintain a log of messages using public operations `log` and `readLast` and some actor private state. This is a list of messages, stored in `stable` (thus persisted), variable `entries`. Its subsequent upgraded version (highlighted inline) records additional time stamps, extends the code for `log` to do so, and adds the public operation `readUntil` to query the log by time. Upgrading the original version preserves the current state of entries while defaulting the times for existing entries to 0. The public interface evolves to a subtype, continuing to satisfy existing clients while offering new functionality to future clients. The retained private state evolves to a (trivial) supertype.

3.1 Stable Variables

A Motoko program uses an actor as top-level component abstraction, containing state and functionality. The variables

```
actor Graph {
 type Node = { // object type
   number : Nat;
   // mutable array of Node references
   var edges: [Node];
 };
 stable var start: Node = …;
 flexible var temporary : Node = …;
}
```

Figure 4. Example Motoko program with two actor variables referring to a graph structure. `start` is retained across upgrades, while `temporary` is reinitialized on an upgrade.

declared inside the actor represent the state and can refer to object structures of arbitrary shape. Therefore, the actor variables serve as a root set for objects allocated in the heap. Motoko is garbage-collected, i.e. objects that are not reachable by actor variables or other temporary references (e.g. in the call stack) are automatically reclaimed with some delay.

Motoko distinguishes between two types of actor variables, see also Figure 4:

- **Stable variables**: Denoted by the keyword `stable`, these variables are persistent across upgrades. This means that all objects that are transitively reachable from stable variables also survive the upgrade, assuming that the new program version still wants to use this state. Therefore, `stable` implies a deep persistence of the referenced object structure, also supporting cycles and sharing.
- **Flexible variables**: Denoted by the keyword `flexible`, these variables are re-initialized on an upgrade. This means that objects reachable from flexible variables can be discarded on upgrade, if not otherwise reachable from stable variables. Flexible variables are useful for transient information, e.g. a recent sales list that is reset on upgrade.

Figure 5 depicts a possible heap structure for the example program. All objects reachable from stable variables (black) constitute stable objects that are persistent across upgrades. All remaining objects reachable from flexible variables (gray) denote transient objects which are temporarily available in the same program version. All white objects are garbage and can be reclaimed.

3.2 Stable Types

Not all values are suited for persistence across upgrades, for example references to function values and future values are difficult to preserve when the program implementation changes. This is because both function values and future values reference local code, unlike all other values. For this purpose, Motoko mildly restricts the types that can be used for stable variables to a well-defined subset of *stable types*,

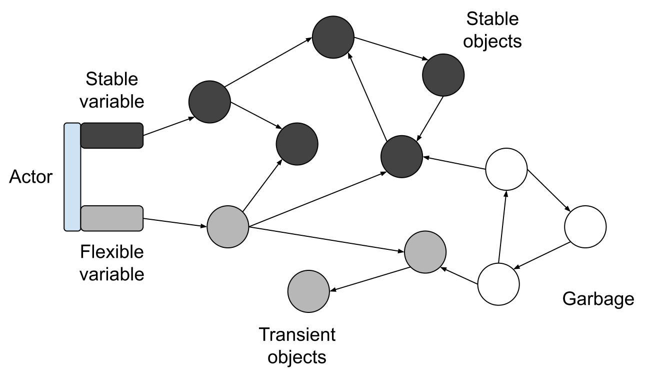


Figure 5. Longevity of objects implied by transitive reachability from actor variables

the recursively defined set of all types excluding function types, future types and constructed types containing them. Actor types and actor functions, referencing remote objects and messaging endpoints, are also stable. The stability of the type of a stable variable is statically checked.

In contrast, flexible variables can be of any type, including function types and futures.

3.3 Upgrades

The code of a Motoko program can be changed to a new version that is deployed to the blockchain as a replacement for a previous version of that program. During this upgrade process, not only does the program logic need to be replaced but also the existing data of the previous versions needs to be migrated to the new version.

In Motoko, the policy for retaining data across program versions is defined declaratively, using `stable` variable declarations. If a stable variable is newly introduced in the upgrade, its initial value is determined by evaluating its defining expression (the initializer). If both the previous version and the next version declare a stable variable with the same name, the value of that variable (including all its reachable state) is retained across the upgrade, ignoring its initializer. Otherwise, if a stable variable is present in the old but absent in the new version, its old value is dropped. The value of a `flexible` variable is always obtained by evaluating its initializer, ignoring any declaration of that variable (stable or not) in the previous version. The conditional initialization of a stable variable, depending on its presence in a previous version (if any), caters for both upgrades as well as the initial deployment of an actor (in which all stable initializers are evaluated).

Figure 6 exemplifies the upgrade process from version 0 to 1. The stable variables `b` and `c` with their reachable state are preserved on upgrade. However, variable `a` is dropped, and variable `d` is freshly initialized. Variables `e` and `f` are not persisted across the upgrade, because they are declared flexible in at least one of the versions.

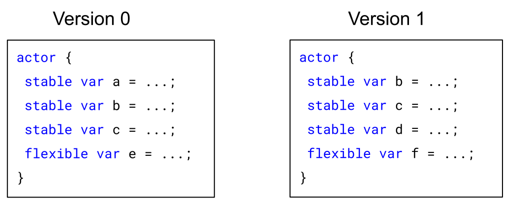

Version 0
```
actor {
 stable var a = ...;
 stable var b = ...;
 stable var c = ...;
 flexible var e = ...;
}
```
Version 1
```
actor {
 stable var b = ...;
 stable var c = ...;
 stable var d = ...;
 flexible var f = ...;
}
```

Figure 6. Upgrade example with different actor variable declarations: Only `b` and `c` are saved across the upgrade.

Generally, the type of a stable variable cannot be arbitrarily changed between program versions but can only evolve to a supertype. This is necessary for the system to implement a defined migration from the original type and representation to the new version. Users have a choice between implicit or explicit migration when evolving data on an upgrade.

3.4 Implicit Migration

Certain data can be automatically migrated to a new program version. This is essentially the case when the type is identical or there exists a defined migration path that is implemented by the runtime system. In Motoko, the runtime system checks that matching stable variables support an implicit migration path, preventing misinterpretation or corruption of data.

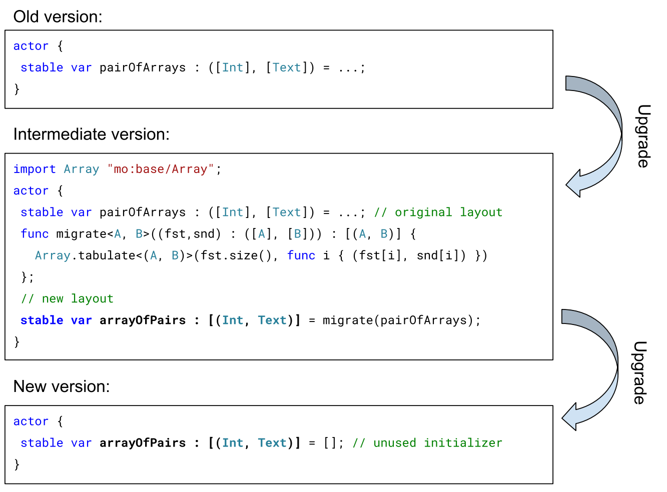
The implicit migration path is determined by the language's permissive subtyping relation: A stable variable can be reused in a next program version if its old type is a subtype of its new type. Subtyping ensures that the old value is compatible with its new type. This allows one to, for example, promote unsigned big integers to signed big integers, remove fields from records or promote their types, add options to variants or promote their types, and even generalize finite types to recursive types. For type soundness, the types of mutable record fields and mutable arrays, that could be aliased, can only change to structurally equivalent types. None of the subtyping changes require a change in representation and are trivial identities on the values, not coercions. For higher flexibility, stable variables can be freely added and their mutability be changed.

Other type changes to stable variables are rejected on an upgrade and require an explicit migration.

3.5 Explicit Migration

There exist many data evolution scenarios that cannot be addressed by implicit migration, and ultimately, only the developers are capable of defining the migration path. For this purpose, there exists an explicit migration approach that allows changing the data representation of a persistent program arbitrarily on an upgrade.

For explicit migrations, developers take a three-step approach:



Old version:

```
actor {
 stable var pairOfArrays : ([Int], [Text]) = ...;
}
```

Intermediate version:

```
import Array "mo:base/Array";
actor {
 stable var pairOfArrays : ([Int], [Text]) = ...; // original layout
 func migrate<A, B>((fst,snd) : ([A], [B])) : [(A, B)] {
   Array.tabulate<(A, B)>(fst.size(), func i { (fst[i], snd[i]) })
 };
 // new layout
 stable var arrayOfPairs : [(Int, Text)] = migrate(pairOfArrays);
}
```

New version:

```
actor {
 stable var arrayOfPairs : [(Int, Text)] = []; // unused initializer
}
```

Figure 7. Complex data evolution with two upgrades

1. Declare a new stable variable with a new data representation and a new variable name, retaining the old stable variable.
2. Initialize the new variable from the old variable (and other state) using custom Motoko logic.
3. Discard the old stable variable, once all data has been migrated to the new variable.

Figure 7 illustrates a more complex migration with custom logic, evolving a pair of arrays to an array of pairs: The intermediate version defines the new stable variable (step 1) and the data migration logic (step 2). The final version drops the old stable variable (step 3). The upgrade skips the initializer of the retained stable variable `arrayOfPairs`.

4 Implementation

Implementing orthogonal persistence in an efficient and safe manner requires a bespoke compiler and runtime system design. In the following sections, we elaborate on the specific design and implementation aspects.

4.1 Blockchain Prerequisites

To enable our compilation of orthogonal persistence, the following IC extensions were needed:

- Retention of main memory across upgrades: We added an option for canisters to retain the Wasm main memory on upgrades instead of discarding it. This option is only used by Motoko which implements orthogonal persistence.
- Passive data segments: In order to avoid static addresses in a program binary, we enabled passive data segments, a feature of the Wasm standard, that allows loading static data content at a dynamic address.
- 64-bit main memory: Originally, the IC only offered 32-bit main memory. In light of increased scalability needs and for supporting large persistent main memory, the Wasm Memory64 feature [21] has been enabled on the IC. With this, orthogonal persistence attains the same scalability as the stable memory that was already 64-bit-based.

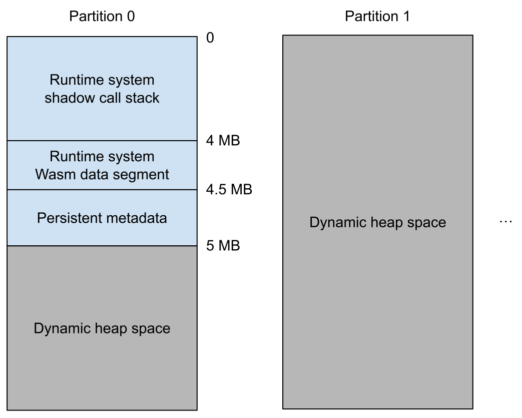


Figure 8. Predefined memory layout

4.2 Memory Layout

In a co-design between the compiler and the runtime system, Motoko's main memory is arranged in a structure that is invariant of the compiled program version, cf. Figure 8: The lower 4 MB are reserved for a dedicated shadow call stack of the Motoko runtime system. The next 512 KB serve as a reserved space for the special Wasm data segment used by the Motoko runtime system (see Section 4.8). In the range between 4.5 MB and 5 MB, persistent metadata is stored. The dynamic heap space begins at 5 MB. Motoko uses a partitioned heap that serves for incremental compacting garbage collection, see Section 4.5.

4.3 Persistent Metadata

The persistent metadata describes all anchor information for the program to resume after an upgrade. More specifically, it comprises the following data:

- A stable heap version that allows evolving the persistent memory layout in the future.
- The stable subset of the main actor, containing all stable variables declared in the main actor.
- A descriptor of the stable static types to check memory compatibility on upgrades.
- The runtime state of the garbage collector, including the dynamic heap metadata and memory statistics.
- Diagnostic information, such as upgrade performance metrics.
- A reserve for future metadata extensions.

4.4 Compatibility Check

Upgrades are only permitted if the new program version is compatible with the old version, such that the runtime system guarantees a compatible memory structure. This check is performed both statically as a precondition of deployment, and dynamically, as part of the upgrade transaction, in case of racing deployments. For the dynamic check, the compiler generates the type descriptor, a type table, that is recorded in the persistent metadata. Upon an upgrade, the new type descriptor is compared against the existing type descriptor: failure of this dynamic subtype check causes the upgrade to roll back.

4.5 Garbage Collection

To efficiently support orthogonal persistence, the Motoko runtime system uses an incremental garbage collector (GC) which relies on a partitioned heap with objects carrying a forwarding pointer [9].

The incremental GC is particularly suited because it is designed to scale to large heaps and the stable heap design also aims to increase scalability.

The garbage collection state needs to be persisted and retained across upgrades. This is because the GC may not yet be completed at the time of an upgrade, with object forwarding still in play. The heap partition metadata is organized as a dynamic list of partition tables, to potentially cover the entire address space.

The garbage collector uses two kinds of root sets:

- **Persistent roots**: These refer to root objects that need to survive program upgrades.
- **Transient roots**: These cover additional roots that are only valid in a specific version of a program and are discarded on an upgrade.

The persistent roots are registered in the persistent metadata and comprise:

- All stable variables of the main actor, to be stored during an upgrade.
- The stable type table (cf. Section 4.4).

The transient roots are referenced by the Wasm data segments and comprise:

- All global variables of the current version, including flexible variables.
- Any continuations awaiting inter-actor calls.
- The constant object pool (cf. Section 4.7).

4.6 Main Actor

On an upgrade, the main actor can be recreated with the existing stable variables being recovered from the persistent roots. The remaining actor variables, the flexible fields as well as new stable variables, need to be (re)initialized.

As a result, the GC can collect unreachable flexible objects of previous canister versions. Unused stable variables of former versions can also be reclaimed by the GC, as well as the contents of stable fields that have been promoted to the top type `Any`. The latter is safe because the language does not support dynamic casts.

4.7 No Static Allocations

Any optimization based on static allocation, such as a static heap, needs to be abandoned. Even statically-known, constant objects have to be allocated in the dynamic heap. This is because these objects may also need to survive upgrades and the persistent main memory cannot accommodate a growing static heap of a new program version in front of the existing dynamic heap. The incremental GC can no longer skip these objects but must examine them and resolve forwarding pointers as for dynamically allocated objects.

For memory and runtime efficiency, object pooling is implemented for compile-time-known constant objects (with side-effect-free initialization). These objects are eagerly created on program initialization/upgrade in the dynamic heap. Any reference to a constant object is looked up whenever its value is needed at runtime. This is done by using a statically known index into a table of references. This table forms part of the transient GC root set.

The runtime system avoids any global Wasm variables for state that needs to be preserved on upgrades. Instead, this runtime state is stored in the persistent metadata.

4.8 Data Segments

Only passive Wasm data segments are used by the Motoko compiler and runtime system. In contrast to ordinary active data segments, passive segments can be explicitly loaded to a dynamic address. This simplifies two aspects:

- The compiled Wasm code can contain arbitrarily large data segments (to the maximum that is supported by the IC). The segments can be loaded to the dynamic heap when needed.
- The IC can simply retain the main memory on an upgrade without needing to patch any active data segments of the new program version to the persistent main memory.

However, more specific handling is required for the runtime system of Motoko that is implemented in Rust: The Rust compiler generates an active data segment. Our linker changes this segment to the passive mode and loads it to the predefined static address of 4 MB (cf. Section 4.2) during the canister initialization and upgrades. The location and size of the runtime system data segments is limited to a defined reserve of 512 KB. This is acceptable because the runtime system only requires a controlled small amount of memory for its data segments, independent of the compiled Motoko program. If there is a future need for larger runtime system static data than 512 KB, this can be supported by using additional passive data segments that are placed at dynamic heap addresses.

4.9 Migration Path of Last Resort

Anticipating that the persistent main memory layout may need to be changed in the future, the runtime system additionally supports a persistence mechanism of last resort, namely by serialization and deserialization of the persistent heap to and from stable memory in a new data format. This format is self-describing and preserves all sharing from the source heap, avoiding the serious limitations of the classical serialization technique.

Arbitrarily large data can be serialized and deserialized beyond the instruction limit of upgrades: Large data serialization and deserialization is split in multiple transactions, running before and/or after the actual blockchain program upgrade to migrate large heaps. Of course, other transactions will be blocked during this process and the necessary upgrade authorization is required to initiate this process. The algorithm is an adaptation of Cheney's graph copying algorithm [12], extended to accommodate a change in representation and incremental execution.

This migration would only occur in rare cases. Using a defined long-term serialization format allows flexible migration compatibility across all compiler versions. This migration path leaves us the option for radical changes in the future, e.g. to introduce a new GC or rearrange the persistent metadata.

4.10 Classical Implementation

Previous versions of Motoko already implemented orthogonal persistence but in a significantly less efficient and robust way. On upgrade, the runtime would first serialize then deserialize the stable roots between main and stable memory. With this approach, developers experienced severe scalability and performance issues, since the serialization / deserialization process is very expensive, often exceeding the instruction limit of the blockchain transaction, ultimately preventing upgrades. Another limiting factor was that the serialization format duplicated shared immutable objects, leading to potential size explosion during serialization. The serialization algorithm was type-directed and recursive, thus prone to data dependent stack overflows, and difficult to scale to larger heaps. With this classic implementation, the applicability of orthogonal persistence was limited to low volumes of data and data of moderate complexity. Removing those limitations was the primary motivation for the new enhanced support of orthogonal persistence.

When migrating from the old serialization-based stabilization to the new persistent heap, the old data is deserialized one last time with the old format from the stable memory and then placed in the new persistent heap layout. Once operating on the persistent heap, the system should prevent downgrade attempts to the old serialization mechanism.

5 Experimental Results

We evaluate Motoko's orthogonal persistence on the Internet Computer in terms of code complexity, program upgrade costs, and general data access performance. For this purpose, the orthogonal persistence is compared against the classical approaches of persistence, namely serializing main memory data to stable memory on an upgrade and using stable data structures that directly access stable memory.

5.1 Benchmark

We assembled a series of application cases for evaluation in three settings:

- **Orthogonal persistence**: The programs are implemented in Motoko by using the programming model and runtime support presented in this paper.
- **Stabilization on upgrade**: The identical Motoko programs are run with a classical mechanism of serializing the relevant data from main memory to stable memory before an upgrade and deserializing the data back to main memory after the upgrade (Section 4.10).
- **Stable data structures**: The programs are analogously implemented in Rust and use stable data structures that directly tunnel accesses to the stable memory.

Table 1 summarizes the application cases in our benchmarks and how the data is mapped in the corresponding programming languages. For repeatable measurements, we always use seeded pseudo-randomization with the same logic in Motoko and Rust [1].

We took care to implement the scenarios as optimal as possible in the corresponding languages: The set of stable data structures in Rust is limited to stable vectors (for lists) and stable B-trees (for maps), while in Motoko, we can use a broader set of data structures in main memory (e.g. an array list, a linked list, or a red-black tree).

5.2 Code Complexity

Naturally, persisting data in stable data structures involves extra programming effort, as needed for the Rust version of our benchmark cases. On the other hand, Motoko only requires the `stable` keyword for the actor fields to enable orthogonal persistence. Figure 9 gives an example of persistence artifacts needed for the `Auction` benchmark. On average, for our benchmark, we counted 17 SLOC for persistence support in Rust and one keyword in Motoko.

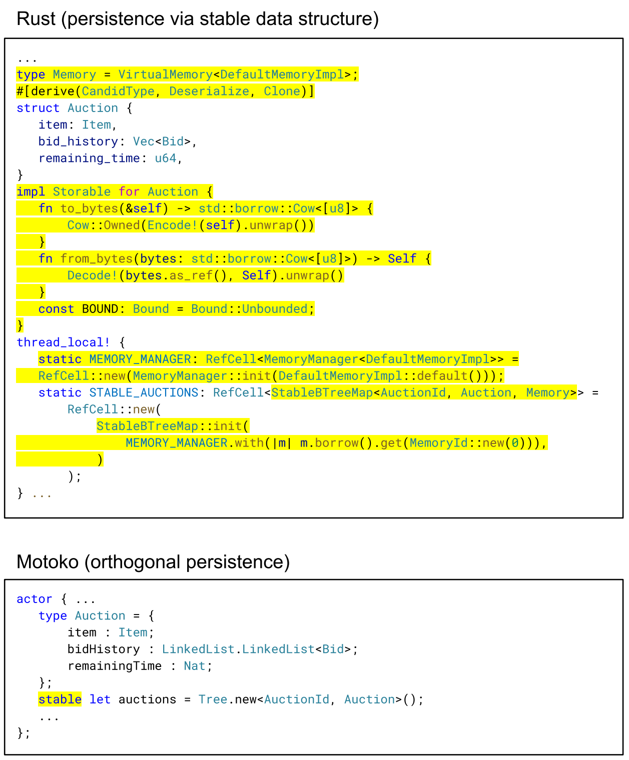
5.3 Upgrade Costs

On the Internet Computer, the runtime costs of an application is determined by the number of executed Wasm instructions, by applying a weight per instruction. Our main motivation for the runtime system design was to allow scalable upgrades that are not depending on the heap size, but

[1]See https://github.com/luc-blaeser/persistence-comparison for the benchmark implementation.

Table 1. Benchmark cases, implemented in Motoko and analogously in Rust

Benchmark	***Description***	***Motoko Design***	***Rust Design***
`List`	List of numbers	Using an array list in main memory	Using the stable vector data structure
`Map`	A key-value store of numbers with random keys	Using a red-black tree in main memory	Using the stable B-tree data structure
`Queue`	A FIFO queue of numbers	Using a linked list in main memory	Inserting entries with an increasing index to a stable B-tree and storing the first and last index.
`Graph`	Random graph with directed edges	Plain objects, each node with a linked list referring to adjacent nodes.	Storing nodes in a stable B-tree by using a node index as key, each node storing the indices of its adjacent nodes.
`Auction`	An auction platform	A red-black tree of auctions containing a linked list of bids.	Stable B-tree of auctions with composed bid list


Rust (persistence via stable data structure)
Motoko (orthogonal persistence)

```
...
type Memory = VirtualMemory<DefaultMemoryImpl>;
#[derive(CandidType, Deserialize, Clone)]
struct Auction {
   item: Item,
   bid_history: Vec<Bid>,
   remaining_time: u64,
}
impl Storable for Auction {
   fn to_bytes(&self) -> std::borrow::Cow<[u8]> {
       Cow::Owned(Encode!(self).unwrap())
   }
   fn from_bytes(bytes: std::borrow::Cow<[u8]>) -> Self {
       Decode!(bytes.as_ref(), Self).unwrap()
   }
   const BOUND: Bound = Bound::Unbounded;
}
thread_local! {
   static MEMORY_MANAGER: RefCell<MemoryManager<DefaultMemoryImpl>> =
   RefCell::new(MemoryManager::init(DefaultMemoryImpl::default()));
   static STABLE_AUCTIONS: RefCell<StableBTreeMap<AuctionId, Auction, Memory>> =
       RefCell::new(
           StableBTreeMap::init(
               MEMORY_MANAGER.with(|m| m.borrow().get(MemoryId::new(0))),
           )
       );
} ...
```

```
actor { ...
   type Auction = {
       item : Item;
       bidHistory : LinkedList.LinkedList<Bid>;
       remainingTime : Nat;
   };
   stable let auctions = Tree.new<AuctionId, Auction>();
   ...
};
```

Figure 9. Highlighting the source code for persistence handling in the `Auction` benchmark.

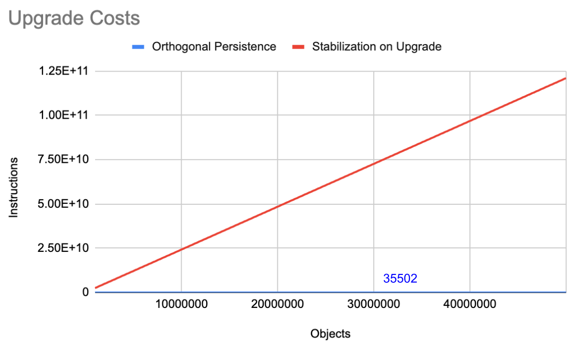


Figure 10. Comparing upgrade costs

Table 2. Benchmark scales for access cost analysis

Benchmark	***Size***
`List`	10 million entries
`Map`	100,000 entries
`Queue`	1 million entries
`Graph`	100,000 nodes with 1 to 5 random edges each
`Auction`	100,000 auctions with 2 bids each

only on the number and complexity of types. We verified this by measuring the number of executed instructions per upgrade for the different benchmark cases by comparing the orthogonal persistence of this paper with the stabilization on upgrade (Section 4.10).

Figure 10 shows the upgrade costs at the example of the auction benchmark with increasing number of objects (and correspondingly, the heap size). As expected, the upgrade costs of the new orthogonal persistence is constant and very low with a few 10 thousand instructions: Only the type-based compatibility check is performed on the upgrade. Conversely, the stabilization costs linearly increase with the amount of objects up to more than 100 billion instructions. On the IC, the maximum number of instructions per upgrade is limited to 200 billion. An upgrade that exceeds the limit will fail and be rolled back, leaving the program in an evolutionary dead end, unable to upgrade.

For the Rust stable data structures, the upgrade cost is also constant and even 10 times lower than with our orthogonal persistence runtime system. This is only because Rust performs no compatibility check on upgrade and is unsafe.

5.4 Data Access Costs

Using stable data structures imposes higher runtime costs because of the accesses that are tunneled to stable memory, also involving serialization and deserialization. To measure the difference, we run the benchmark cases with the application sizes defined in Table 2.

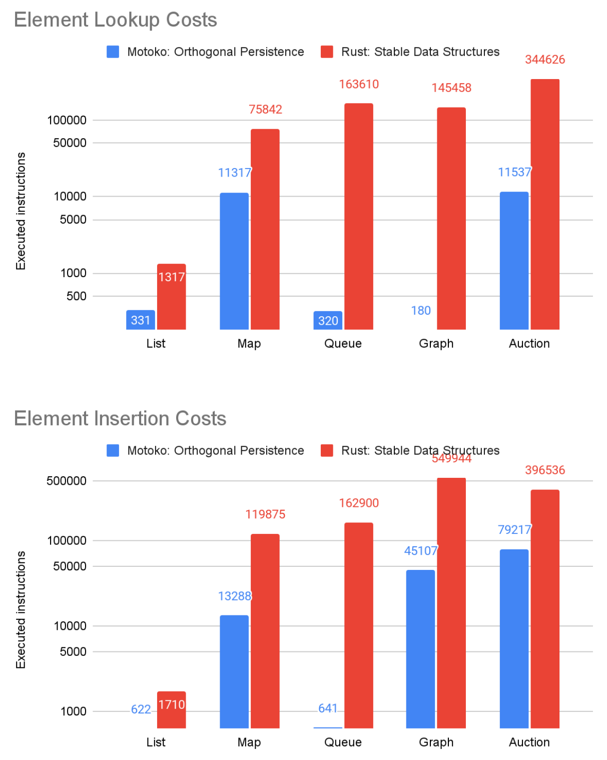


Figure 11. Comparing access costs

Figure 11 compares the average runtime costs for an element lookup and insertion on a logarithmic scale. The costs are again quantified as executed weighted Wasm instructions on the IC. While Motoko can operate directly on main memory, the Rust code must operate indirectly, on stable memory.

As can be seen, the performance costs are orders of magnitudes higher in Rust. This arises for various reasons: (1) The choice of stable data structures is limited, e.g. only a B-tree is available as a map data structure, while Motoko can flexibly choose any data structure that fits the application case best. (2) Serialization and deserialization is involved with each insertion and lookup. (3) Some extra mapping complexity arises with stable memory, e.g. no direct navigation via pointers is possible. Instead, objects need to be looked up via an explicit index in a map.

6 Related Work

Orthogonal persistence has been originally proposed in [3, 5] and has been implemented for Java [6, 7] and other programming languages [4, 8, 24, 25], and also for operating systems [15]. Unfortunately, the concept did not achieve broad adoption in practice. One often mentioned concern is that program structures are bound to the program binary and data evolution is challenging: However, if the runtime system for orthogonal persistence features a powerful enough data evolution facility, data can be flexibly and safely migrated to a new program version [7, 8], similar to database tooling.

Related to orthogonal persistence, dynamic software updating [1, 23, 27] (DSU) implements mechanisms for altering a running program. Assuming DSU is supported on top of a persistent memory, it can realize the same properties as our implementation: A dynamic update would need to guarantee safety and support a combination of synthesized and manual state transformers (for implicit and explicit migrations). The point between blockchain transactions can be viewed as the safe point for a dynamic software update. Like DSU, our fast upgrades offer high availability, but without the flexibility (and complexity) of fine-grained binary patching.

Currently, for most programming languages used on a blockchain, program upgrades remain a cumbersome and fragile process. Ethereum necessitates a redirection architecture with multiple smart contracts [17, 26] to deploy program changes. A common pattern is to store the data inside the proxy, limiting data migration possibilities quite considerably. Other blockchains require extra programming effort by way of a specific API [14, 28] or library [19] to preserve any long-term state across upgrades.

7 Conclusion

Program upgrades are a dominant software engineering aspect when developing smart contracts or other blockchain applications, involving substantial complexity, overheads, and error-proneness. Programmers typically have to prepare upgrades by complicated redirection architectures, or by explicitly storing program data to a secondary memory or in special data structures. We overcome this complexity by featuring orthogonal persistence as part of Motoko, a programming language that is specifically designed for the Internet Computer blockchain: Developers can conveniently program any data structures of first-order types in the standard Motoko language concepts, without needing to use any API or library. The runtime system automatically persists the necessary objects and guides safe, fast, and flexible program upgrades, while supporting both implicit and explicit data migration paths.

Availability

The Motoko programming language is open source and available at https://github.com/dfinity/motoko.

Acknowledgment

Andreas Rossberg is the main designer of the Motoko language. He deserves credit for this well thought out and elegant language. Various people provided valuable feedback and support in realizing this work: Ulan Degenbaev, Dimitris Sarlis, Alexandru Uta, Yan Chen, Islam El-Ashi, Martin Rasyzk, Björn Tackmann, Samuel Burri, Jan Camenisch, and Dominic Williams. We owe our gratitude to all of them.

References

[1] Babiker Hussien Ahmed, Sai Peck Lee, Moon Ting Su, and Abubakar Zakari. 2020. Dynamic software updating: a systematic mapping study. *IET Software* 14, 5 (2020), 468–481.

[2] Maksym Arutyunyan, Andriy Berestovskyy, Adam Bratschi-Kaye, Ulan Degenbaev, Manu Drijvers, Islam El-Ashi, Stefan Kaestle, Roman Kashitsyn, Maciej Kot, Yvonne-Anne Pignolet, Rostislav Rumenov, Dimitris Sarlis, Alin Sinpalean, Alexandru Uta, Bogdan Warinschi, and Alexandra Zapuc. 2023. Decentralized and Stateful Serverless Computing on the Internet Computer Blockchain. In *Proceedings of the 2023 USENIX Annual Technical Conference* (Boston, MA, USA) *(ATC ’23)*.

[3] MP Atkinson. 1978. Database systems and programming languages. In *Proceedings of 4th VLDB Conference*. 408–419.

[4] Malcolm Atkinson, Ken Chisholm, and Paul Cockshott. 1982. PS-algol: an algol with a persistent heap. *SIGPLAN Not.* 17, 7 (jul 1982), 24–31. https://doi.org/10.1145/988376.988378

[5] Malcolm Atkinson and Ronald Morrison. 1995. Orthogonally persistent object systems. *The VLDB Journal* 4, 3 (jul 1995), 319–402.

[6] M. P. Atkinson, L. Daynès, M. J. Jordan, T. Printezis, and S. Spence. 1996. An orthogonally persistent Java. *SIGMOD Rec.* 25, 4 (dec 1996), 68–75. https://doi.org/10.1145/245882.245905

[7] Malcolm P. Atkinson and Mick J. Jordan. 2000. A review of the rationale and architectures of PJama - a durable, flexible, evolvable and scalable orthogonally persistent programming platform. In *SMLI TR*. https://labs.oracle.com/pls/apex/f?p=94065:10:11580106517824:1780

[8] Luc Bläser. 2007. Persistent Oberon: A Programming Language with Integrated Persistence. In *Asian Symposium on Programming Languages and Systems*. Springer, 71–85.

[9] Luc Bläser, Claudio Russo, Ulan Degenbaev, Ömer S. Ağacan, Gabor Greif, and Jason Ibrahim. 2023. Collecting Garbage on the Blockchain. In *Proceedings of the 15th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages* (Cascais, Portugal) *(VMIL 2023)*. Association for Computing Machinery, New York, NY, USA, 50–60. https://doi.org/10.1145/3623507.3627672

[10] Vitalik Buterin. 2024. *Ethereum Whitepaper*. https://ethereum.org/en/whitepaper

[11] Jan Camenisch, Manu Drijvers, Timo Hanke, Yvonne-Anne Pignolet, Victor Shoup, and Dominic Williams. 2022. Internet Computer Consensus. In *Proceedings of the 2022 ACM Symposium on Principles of Distributed Computing* (Salerno, Italy) *(PODC’22)*. Association for Computing Machinery, New York, NY, USA, 81–91. https://doi.org/10.1145/3519270.3538430

[12] C. J. Cheney. 1970. A nonrecursive list compacting algorithm. *Commun. ACM* 13, 11 (nov 1970), 677–678. https://doi.org/10.1145/362790.362798

[13] The Internet Computer. 2024. *The Internet Computer Interface Specification. IC Management Canister: Code Upgrade*. https://internetcomputer.org/docs/current/references/ic-interface-spec#ic-management-canister-code-upgrade

[14] The Internet Computer. 2024. *The Internet Computer Interface Specification. Stable Memory*. https://internetcomputer.org/docs/current/references/ic-interface-spec#system-api-stable-memory

[15] Alan Dearle, Rex Di Bona, James Farrow, Frans Henskens, Anders Lindström, John Rosenberg, Francis Vaughan, et al. 1994. Grasshopper: An orthogonally persistent operating system. *Computing Systems* 7, 3 (1994), 289–312.

[16] Alan Dearle, Graham N. C. Kirby, and Ron Morrison. 2010. Orthogonal Persistence Revisited. In *Object Databases*, Moira C. Norrie and Michael Grossniklaus (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–22.

[17] Ethereum. 2024. *Upgrading Smart Contracts*. https://ethereum.org/en/developers/docs/smart-contracts/upgrading

[18] DFINITY Foundation. 2024. *The Motoko Programming Language*. https://github.com/dfinity/motoko

[19] DFINITY Foundation. 2024. *Stable Data Structures for Rust*. https://github.com/dfinity/stable-structures

[20] WebAssembly Community Group. 2022. *WebAssembly Specification, Version 2.0*. https://webassembly.org/

[21] WebAssembly Community Group. 2024. *WebAssembly. Memory64 Proposal*. https://github.com/WebAssembly/memory64/blob/master/proposals/memory64/Overview.md

[22] Carl Hewitt, Peter Bishop, and Richard Steiger. 1973. A universal modular ACTOR formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence* (Stanford, USA) *(IJCAI’73)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 235–245.

[23] Michael Hicks and Scott Nettles. 2005. Dynamic software updating. *ACM Trans. Program. Lang. Syst.* 27, 6 (nov 2005), 1049–1096. https://doi.org/10.1145/1108970.1108971

[24] Ted Kaehler and Glenn Krasner. 1983. LOOM—large object-oriented memory for Smalltalk-80 systems. *Smalltalk-80: Bits of History, Words of Advice* (1983), 251–270.

[25] David C.J. Matthews. 1987. *A persistent storage system for Poly and ML*. Technical Report UCAM-CL-TR-102. University of Cambridge, Computer Laboratory. https://doi.org/10.48456/tr-102

[26] OpenZeppelin. 2024. *Upgrading Smart Contracts*. https://docs.openzeppelin.com/learn/upgrading-smart-contracts

[27] Habib Seifzadeh, Hassan Abolhassani, and Mohsen Sadighi Moshkenani. 2013. A survey of dynamic software updating. *Journal of Software: Evolution and Process* 25, 5 (2013), 535–568.

[28] Adam Welc and Sam Blackshear. 2023. Sui Move: Modern Blockchain Programming with Objects. In *Companion Proceedings of the 2023 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity* (Cascais, Portugal) *(SPLASH 2023)*. Association for Computing Machinery, New York, NY, USA, 53–55. https://doi.org/10.1145/3618305.3623605

Received 2024-07-11; accepted 2024-08-21