Persistent Software Transactional Memory in Haskell

NICOLAS KRAUTER* and PATRICK RAAF*, Johannes Gutenberg University, Germany
PETER BRAAM, University of Oxford, United Kingdom
REZA SALKHORDEH, Johannes Gutenberg University, Germany
SEBASTIAN ERDWEG, Johannes Gutenberg University, Germany
ANDRÉ BRINKMANN, Johannes Gutenberg University, Germany

Emerging persistent memory in commodity hardware allows byte-granular accesses to persistent state at memory speeds. However, to prevent inconsistent state in persistent memory due to unexpected system failures, different write-semantics are required compared to volatile memory. Transaction-based library solutions for persistent memory facilitate the atomic modification of persistent data in languages where memory is explicitly managed by the programmer, such as C/C++. For languages that provide extended capabilities like automatic memory management, a more native integration into the language is needed to maintain the high level of memory abstraction. It is shown in this paper how persistent software transactional memory (PSTM) can be tightly integrated into the runtime system of Haskell to atomically manage values of persistent transactional data types. PSTM has a clear interface and semantics extending that of software transactional memory (STM). Its integration with the language's memory management retains features like garbage collection and allocation strategies, and is fully compatible with Haskell's lazy execution model. Our PSTM implementation demonstrates competitive performance with low level libraries and trivial portability of existing STM libraries to PSTM. The implementation allows further interesting use cases, such as persistent memoization and persistent Haskell expressions.

CCS Concepts: • **Hardware → Non-volatile memory**; • **Software and its engineering → Runtime environments**; **Concurrency control**; **Multithreading**; *Memory management*; *Garbage collection*; *Allocation / deallocation strategies.*

Additional Key Words and Phrases: persistent transaction support, non-volatile heap

ACM Reference Format:
Nicolas Krauter, Patrick Raaf, Peter Braam, Reza Salkhordeh, Sebastian Erdweg, and André Brinkmann. 2021. Persistent Software Transactional Memory in Haskell. *Proc. ACM Program. Lang.* 5, ICFP, Article 63 (August 2021), 29 pages. https://doi.org/10.1145/3473568

1 INTRODUCTION

Most applications need to protect their internal state against data corruption. For example, a financial application has to ensure that all acknowledged transactions of their customers are correctly reflected in the corresponding balances even in case of a system crash. However, such a crash results in irrevocable data loss if the application only resides in volatile memory and persistent

*Both authors contributed equally to this research.

Authors' addresses: Nicolas Krauter; Patrick Raaf, Johannes Gutenberg University, Mainz, Germany, n.krauter@uni-mainz.de, raaf@uni-mainz.de; Peter Braam, University of Oxford, Oxford, United Kingdom, peter@braam.io; Reza Salkhordeh, Johannes Gutenberg University, Mainz, Germany, rsalkhor@uni-mainz.de; Sebastian Erdweg, Johannes Gutenberg University, Mainz, Germany, erdweg@uni-mainz.de; André Brinkmann, Johannes Gutenberg University, Mainz, Germany, brinkmann@uni-mainz.de.

2475-1421/2021/8-ART63
https://doi.org/10.1145/3473568

storage therefore has to be used to enable fail-safe execution. Persistent storage has been accessed, until recently, mostly through the I/O path of the operating system which has high latencies and primarily offers block granular access [Breuer 2003] for logging and synchronization.

Emerging persistent memory (PM) in commodity hardware is byte-addressable, offers latencies comparable to DRAM, and storage capacities that can significantly exceed DRAM. Persistent memory can be transparently integrated into existing software architectures to protect application state through the file system interface [Moti et al. 2021; Rao et al. 2014; Xu and Swanson 2016]. However, the full potential of PM can only be leveraged if applications directly access and modify persistent data in-place so that data does not have to be synchronized between the volatile and the persistence domain [Rudoff 2017]. Direct access to persistent state requires different write-semantics compared to volatile memory to be able to recover from system crashes. Programmers have to support power fail-safe atomic updates, correctly manage the interplay between volatile and persistent memory, and steer cache flushes. In the last decade, a large number of libraries have been developed to alleviate the unique challenges of PM [Brown and Avni 2016; Coburn et al. 2011; Liu et al. 2017; Rudoff 2017; Volos et al. 2011]. These libraries offer suitable abstractions for languages like C/C++, which support direct memory operations. However, library-based approaches are less applicable in languages providing automatic memory management.

In languages with automatic memory management, PM libraries cannot modify the underlying runtime system to support PM and thus must implement their own separate memory model [Wu et al. 2018]. Library users therefore need to explicitly identify and manage objects and data structures that reside in PM and ensure their referential integrity. In contrast, the direct integration with the language can sustain the original memory abstraction for PM, effectively reducing the burden on the programmer. While considerable effort has been taken for imperative languages like Java [Shull et al. 2019; Wu et al. 2020, 2018], the integration of PM into purely functional programming languages has not yet been investigated.

This paper focuses on the lazy, purely functional programming language Haskell [Hudak et al. 1992] and evaluates the idiomatic integration of PM into the Haskell language runtime. Transactions offer proper semantics to express consistent state transitions [Haerder and Reuter 1983] and are therefore used by many frameworks and library abstractions for PM. Haskell is an ideal candidate for integration of persistent transactions as it already offers a well-accepted implementation of software transactional memory (STM) in its runtime system [Harris et al. 2005] that enables the concurrency-safe modification of memory contents. Haskell's focus on pure functions limits the scope of potential in-place modifications to specific mutable memory objects, even in its runtime implementation. These can be controlled efficiently by individually tailored lightweight logging solutions. Moreover, expressions of Haskell applications are represented in memory as a graph that contains functions as well as data, which is reduced lazily in a call-by-need fashion. By automatically persisting computations, execution state can be transparently shared across runs.

We have extended STM transactions and their semantics in Haskell to provide atomicity, consistency, isolation and durability, i.e. the well-known ACID guarantees, for PM. Our persistent software transactional memory (PSTM) solution enables composable transactions in which persistent Haskell variables can be created or modified. PSTM offers the same concurrency guarantees as STM, provides almost the same interface as STM, and provides easy migration of STM code to PSTM. PSTM ensures the recoverability of transactions in case of power-failures by correctly ordering cache flushes to the persistence domain and lightweight logging of pointer updates of persistent variables. Our solution is fully compatible with Haskell's lazy execution model and PSTM can be used to store arbitrary expressions in PM. In particular, unevaluated function applications stored in PM are efficiently reduced on demand and retain their execution state across runs. This

leads to interesting capabilities like persistent memoization and inherently restartable persistent subprograms.

Our PSTM solution is fully integrated into GHC's runtime system, as it needs to *understand* the difference between volatile DRAM and PM to support Haskell's execution model as a first principle. We have designed a hybrid heap where PM is provided as an additional, automatically managed heap region. Memory allocation and garbage collection have been adapted within the PM region in a way that the lifetime of objects must not be explicitly handled by the programmer but is defined by their reachability.

Our performance evaluation of PSTM shows a slowdown of persistent transactions compared to a volatile STM implementation between $1.3x$ and $2.8x$ on the same memory technology. This slowdown slightly increases when PSTM is running on today's Intel Optane PM, which still has increased access latencies compared to DRAM. A comparison with persistent C/C++ libraries, i.e. Mnemosyne [Volos et al. 2011], PMDK [Rudoff 2017], Romolus [Correia et al. 2018] and OneFile [Ramalhete et al. 2019], shows that the optimistic synchronization of PSTM outperforms dedicated persistent libraries in the presence of many concurrent write transactions. Using two common memoization libraries, we compared the performance of persistent graph reduction with their volatile evaluation.

Although our approach specifically targets the integration into the Haskell language, our concepts can be applied to other (functional) programming languages. Limited mutability of well-defined memory structures and reachability-based garbage collection can significantly reduce (transaction) logging overheads and effectively allows to hide the specifics of PM from the programmer, e.g. by automatically persisting related objects in a self-contained manner. Graph-based lazy execution models can be adapted to automatically persist computation state across runs using only atomic pointer updates.

In summary, our work makes the following contributions:

- Transparent PSTM extension of Haskell's STM mechanism to provide power-fail safe persistence while maintaining composability
- Integration of PM into Haskell's heap design to enable the automatic memory management of persistent structures
- Extending Haskell's laziness capabilities to PM
- Exemplary adaptation of existing libraries to PSTM and integration of persistent memoization
- Performance evaluation and comparison of PSTM

The remaining paper is structured as follows: Section 2 presents the technical and methodological background of this work. Section 3 discusses the design of the PSTM API and Section 4 presents its implementation. Section 5 presents the results, including performance measurements of PSTM and a comparison with library-based approaches. A summary is given in Section 6.

2 RELATED WORK AND BACKGROUND

We will introduce a Haskell language-level abstraction to interact with PM. However, the development of transparent *persistent programming* abstractions to interact with non-volatile storage technologies has a long history of active research, which is described in the following section. Subsequently, we discuss the peculiarities of PM as well as related work aiming to simplify its usage. As our solution is integrated into the runtime system of the Glasgow Haskell Compiler [Hall et al. 1992], we will give a basic overview of Haskell memory management and the principles of the transaction mechanism STM.

2.1 Persistent Programming

Programs that need to access persistent data mostly rely on dedicated I/O instructions that move data between the persistent media and their volatile in-memory workspace. Explicitly interacting with storage mechanisms outside the control of the application might result in hard to maintain code bases, as correct handling of interfacing, typing, and concurrency have to be resolved. Persistent programming languages can offer a much higher abstraction when interacting with persistent data [Atkinson et al. 1982; Bläser 2007; Hosking and Chen 1999; Morrison et al. 2000]. They often rely on the concept of orthogonal persistence introduced by Atkinson et al. [1983] and further described in more detail by Atkinson and Morrison [1995]. It was derived from the language design principles of correspondence, abstraction, and data type completeness by Tennent [1977] and Strachey [2000] to offer the transparent integration of persistence within the language. Orthogonal persistence is defined by three key principles:

(1) The principle of persistence independence
(2) The principle of data type orthogonality
(3) The principle of persistence identification

Persistence independence relieves the programmer from explicitly managing data movement. Instead, data is moved automatically between transient and persistent storage when needed and the interaction with persistent data resembles that of volatile data. *Data type orthogonality* means that there should be no distinct types for volatile and persistent data to ensure compatibility. *Persistence identification* relates to the ability of the system to identify objects that should be kept persistent. Objects should survive as long as they can be referred to by the application. Usually, these are identified by the reachability from a set of persistent roots. However, the concept of orthogonal persistence is only an abstraction for the programmer. Internally, all data that needs to outlast possible power-failures needs to be explicitly moved through the I/O path. For this, data often needs to be transformed in such a way that consecutive runs can still interpret it. For example, pointer swizzling [Kemper and Kossmann 1993], i.e. converting direct memory references to stable identifiers, needs to be performed. This process is called serialization and needs to be designed in a way suitable for the data structures.

Persistent programming techniques were also investigated in functional programming languages, e.g. by Harper [1985] who introduced a persistent heap to Standard ML. McNally and Davie [1991] modelled a language that combines lazy functional programming with orthogonal persistence to remember executed computations persistently. Quintela and Sánchez [2001] presented an orthogonal persistent implementation of Haskell also supporting laziness. Although orthogonal persistence allows well-formed abstractions to interact with persistent storage, it does not necessary guarantee consistency in terms of atomicity of interrelated operations or concurrency. Marquez et al. [2000] have shown for Java that this issue can be resolved when orthogonal persistence is combined with ACID transactions.

2.2 Persistent Memory

The term persistent memory covers a set of hardware technologies which offer storage-like persistence and memory-like performance [Freitas and Wilcke 2008]. Presently, PM is provided in form of DIMMs that reside on the memory bus. This allows the persistent modification of memory using byte granular load/store instructions.

Various PM technologies are under development, including phase-change memory [Raoux et al. 2008], spin-torque RAM [Kultursay et al. 2013], racetrack memory [Parkin et al. 2008], battery-backed DRAM solutions, or Intel's Optane DC persistent memory [Hady et al. 2017]. These solutions differ in density and performance and cover different application areas. The widely available Intel

Optane DC technology targets the server market and enables higher memory densities than DRAM but its access latencies are also increased, e.g., by a factor of 4x for reads [Weiland et al. 2019].

Application-layer access to PM can be established via the I/O path using a file system or by directly mapping PM into the virtual address space of applications [Rudoff 2017]. The file system approach offers the standard OS file operations and the consistency guarantees of the file system apply [Dulloor et al. 2014; Moti et al. 2021; Xu et al. 2017]. Alternatively, mapping PM into the virtual memory allows applications to directly use load and store instructions to access data with no page cache involved and thus no further synchronization required. While this offers the fastest possible access to PM, it also requires the application itself to ensure consistency, e.g., by correctly flushing transient CPU caches to the persistence domain.

Intel's Persistent Memory Development Kit (PMDK) [Rudoff 2017] has been created to simplify application development for PM. The low-level library libpmem within PMDK maps persistent files into an application's address space and provides low-level operations that follow the write semantics of PM. Programmers can develop optimized PM algorithms which use explicit cache flushes or memory fences without relying on hardware specific assembler instructions. Current CPU architectures only support atomic writes up to the size of 8 bytes. Consistent modifications of larger regions in PM therefore require additional techniques like logging. libpmemobj works on top of libpmem and enables transactions that support arbitrary large atomic updates of multiple structures by maintaining an undo log. As the virtual address mapping can change across runs traditional pointers might become invalid. libpmemobj preserves the referential integrity of data in PM by 128-bit persistent pointers with position-independent offsets that are translated at runtime.

Other libraries operate at a similar abstraction level and offer scalability by integrating STM-based transaction mechanisms [Felber et al. 2008; Gottschlich and Connors 2007; Herlihy et al. 2003; Saha et al. 2006]. The C library Mnemosyne [Volos et al. 2011] enables fine grained PM transactions by relying on tinySTM [Felber et al. 2008] for concurrency control. OneFile uses a redo-log that utilizes dual word compare-and-swap (CAS) operations [Ramalhete et al. 2019]. The write-sets of transactions are shared across threads, so that they can be processed cooperatively. Romulus allows consistent modifications by using two replicas of each data structure [Correia et al. 2018]. Modifications are performed in place on the first replica and are synchronized on commit with the second. OneFile and Romulus only control memory accesses to variables wrapped in a specific persistent class. However, they still require to explicitly allocate objects which should reside in PM. Furthermore, all referenced structures from a persistent class have to be persistent to be recoverable in case of a crash. Language-level abstractions like AutoPersist in Java by Shull et al. [2019] significantly reduce the burden for the programmer. They leave the responsibility for moving interrelated objects to PM to the runtime system, so that a self-contained state is automatically ensured. AutoPersist and the C++ library NV-Heaps [Coburn et al. 2011] automatically garbage collect data structures based on their reachability, effectively preventing the programmer from persistent memory space leaks.

2.3 Haskell and Its Memory Management

Haskell is a functional programming language whose expressions operate on immutable data by transforming data to new data. This is in contrast to the imperative instruction set of modern processors which use statements to mutate data in-place. It is therefore the responsibility of the Haskell compiler and runtime system to map the language's execution model to the actual architectures. We here focus on the popular Glasgow Haskell Compiler (GHC) [Hall et al. 1992].

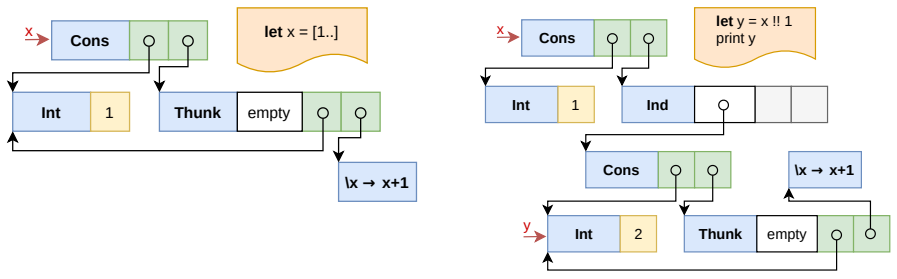
2.3.1 Heap Layout. GHC represents a Haskell program as a graph in the heap that is transformed during its execution by the abstract Spineless Tagless G-Machine, a graph reduction machine that

consists of a set of registers, a stack, and a heap [Peyton Jones 1992]. Haskell uses a non-strict semantic and the graph contains data as well as unevaluated function applications, i.e. computations, which can be lazily reduced as needed. While there are many different types of graph objects the most important types for our discussion are:

- **Constructors** correspond to values of immutable Haskell data constructors. They are stored in weak-head normal form (WHNF), which means that they cannot be reduced further as an outermost expression, but could still contain references to unevaluated expressions inside the graph.
- **Functions** represent Haskell functions and their environment, i.e. their free variables, which are variables not bound to an input argument. The variables bound to arguments are applied explicitly via the stack.
- **Thunks** resemble reducible function applications. Evaluation yields the corresponding result in WHNF. Thunks are replaced by **indirections** that refer to the result. During evaluation, they might be replaced by **blackholes**, which prevent other threads from evaluating the same thunk again.

Heap objects in Haskell, called closures, have a standardized memory layout independent from the stored data [GHC Team 2020b]. A closure contains data and references to other closures in a payload section. Its structure is described in the closures info table. Each closure points to a block of code associated with it, which is called entry code. For thunks this code describes the sequence of instructions needed for their evaluation. The evaluation might result in the allocation of new closures. Info tables and the entry code are shared amongst closures of the same type. They are generated at compile time and reside in the binary which is mapped at runtime. This static region also contains closures for expressions known at compile time which do not require dynamic allocation, such as top level definitions. However, they share the layout of heap objects.

Figure 1a shows a simplified example of a Haskell expression and its representation in memory. Here $x = [1..]$ is the infinite list with only the first element, 1, being evaluated. The list consists of all previously described closure types. The head of the list is a constructor that points to the first element as well as a thunk, i.e. the unevaluated tail of the list. The thunk points to the function $\lambda x \rightarrow x + 1$ that is used during evaluation to create a new list element with the predecessor as argument. Figure 1b shows the list after the evaluation of the next item by forcing y. The thunk is replaced by an indirection to the result, a new constructor pointing to the element as well as to a new thunk representing the new tail of the list. It can be observed that y evaluates to a reference to the list element 2 instead of generating a new closure. This concept, called *sharing*, is heavily used in GHC and possible because of the immutability of values.



(a) Partially evaluated infinite list

(b) List after forcing the next item

Fig. 1. Example of GHC's memory representation and lazy graph reduction on an infinite list

2.3.2 Allocation and Garbage Collection. During the reduction of the closure graph new closures are created while others become unreachable. GHC's memory management subsystem provides functionality for closure allocation and recovers unused memory through a garbage collector. Allocation is realized through a layered approach. Fixed-sized megablocks are allocated from the operating system. Megablocks are internally divided in smaller regions, called blocks. Blocks are described by metadata residing at the beginning of the megablock [GHC Team 2020a]. A Haskell program is decoupled from these allocators; closures are created using a bump allocation strategy in a per-thread region called nursery. It is provided by the runtime system as a contiguous region of blocks where memory is claimed from within the entry code by simply incrementing a pointer.

GHC uses a stop-the-world generational copying garbage collector to reclaim unreachable closures. It traverses the heap starting from a set of roots of the generation that should be collected and retains all reachable closures by moving them to a newly allocated memory region. Afterwards, the blocks of the old one are reclaimed [Marlow et al. 2008]. In contrast, the recently introduced Alligator collector by Gamari and Dietz [2020] inspired by the work of Ueno and Ohori [2016] uses a mark-and-sweep based collection strategy, which allows to reclaim occupied space on a per-closure basis concurrently to the mutator. It is integrated cooperatively with the regular copying collector and can be chosen as an alternative collection strategy for the oldest generation. Reachable closures are retained in place, significantly reducing copy overheads for the most long-living objects. Alligator provides an own allocator for its non-moving heap on top of the block layer. It divides the memory region into segments with fixed-sized sub-blocks. A sub-block stores exactly one closure and differently sized closures are supported by segments with different block sizes.

2.3.3 Software Transactional Memory. Haskell focuses on immutable data structures, whereas some problems are more easily expressed when using mutable state. To revisit the example from the introduction of a financial application, the account balances might be stored as mutable state. Threads processing transfers concurrently need to be isolated from each other to ensure consistency.

STM enables programmers to express consistent state transitions through transactions [Herlihy and Moss 1993]. If a thread needs to modify a set of variables atomically, the modifications are declared in a transaction block. From the view of all other threads these changes happen atomically.

```
data TVar a
newTVar    ::       a → STM (TVar a)
readTVar   :: TVar a → STM a
writeTVar  :: TVar a → a → STM ()
atomically :: STM  a → IO a
```

Listing 1. Excerpt of Haskell's STM API

Haskell's implementation of STM provides the high level API shown in Listing 1 [Peyton Jones et al. 1996]. The mutability is limited to transactional variables called TVars. STM provides its own monad, which allows the composition of operations that create, read, or modify TVars. Additionally, it only allows revertible actions. This is for the reason that a transaction might try to commit multiple times because used variables are changed by other threads in the meantime. STM can only undo modifications to TVars, so the scope of STM is limited to these changes. This excludes operations such as printing to a screen or storing some data on disk. `atomically` is used to perform a series of STM actions atomically with respect to concurrency. The account example from before can be implemented as shown in Listing 2. An account with an initial balance is created using `createAccount`. Funds are moved between accounts with `transfer`.

GHC's runtime provides the mechanisms to execute transactions in an atomic manner. A transaction is first executed in thread local state by maintaining a local log containing the transactions read and

```
type Acc = TVar Int

createAccount :: Int → IO Acc
createAccount b = atomically $ newTVar b

transfer :: Acc → Acc → Int → IO ()
transfer a1 a2 amount = atomically $ do
                            b1 <- readTVar a1
                            b2 <- readTVar a2
                            writeTVar a1 (b1 - amount)
                            writeTVar a2 (b2 + amount)
```

Listing 2. Example of a simple STM transaction transfering *amount* between accounts $a1$ and $a2$

write set. When either `readTVar` or `writeTVar` are called, an entry is added that tracks the current as well as the potential new value of the variable. Finally, STM tries to commit the transaction in order to update the global state. The involved variables are locked and their global current value is checked to match the expected value. When no conflicts are detected, the global TVars are updated and the locks are released. In `createAccount` no conflicts can occur as only new, thread-local TVars are created. When `transfer` transactions are executed concurrently, conflicts are detected in the validation phase. A transaction is restarted if the global state has changed during thread-local execution.

Libraries such as TCache [Corona 2017] allow STM transactions to interact with persistent data. Transactions operate on a global volatile cache that is synchronized with a persistent backend. However, TCache has not been designed for byte-addressable memory and it relies on serialization. This restricts the possible values which can be stored in the backend and limits the lazy evaluation capabilities of Haskell. TCache requires an explicit name for every persistent variable to allow identification across runs. This also requires the adaptation of applications to explicitly interact with persistent state, e.g. by managing the lifetime.

3 DESIGN GOALS AND HIGH-LEVEL INTERFACE

Compared to languages that allow explicit memory modifications, the high abstraction of functional programming languages from the underlying memory requires a different approach to integrate byte-addressable PM idiomatically. PM allows the direct modification of memory contents from within the application without the need for any synchronization between the transient working space and the persistent backend. Haskell nevertheless enforces the purity of values and has no assignment operator to directly modify data. In this section we will present the design of PSTM and its interface which aim to provide a concise abstraction of transactions to consistently interact with PM while relieving programmers from HW-related details. In our design, PSTM serves as the main programming model to persist and retrieve Haskell expressions and its relation to orthogonal persistence will be described subsequently.

3.1 Key Design Principles

We identified the following key principles (KPs) during the design of this high-level abstraction:

(1) **Composability:** Persistent transactions should be composable with other persistent and volatile transactions.
(2) **Direct Access:** PM's byte-addressable access semantics should be leveraged to avoid unnecessary synchronization between volatile and persistent memory.

(3) **Power-Fail Safety:** Persistent data stored in PM should always be recoverable. PSTM needs to preserve two invariants at any point in time:
   (i) Modifications of data in the persistent region of the heap always ensure consistent state transitions.
   (ii) There are no unrecoverable pointers from PM to the volatile heap, as they lose their meaning across runs.
(4) **Identification:** Persistent data needs to be retrievable in consecutive runs by providing a possibility of discovery.
(5) **Lifetime:** An automatic memory management should preserve persisted data across runs while memory for structures which are no longer reachable should be automatically reclaimed.
(6) **Safety:** The PSTM abstraction should allow to express consistent state transitions while reducing the potential of programming errors.
(7) **Compatibility:** The native execution model of Haskell should be preserved as much as possible.
(8) **Security:** Persistent data has to be protected against unauthorized access and modification.

3.2 Persistent Transaction Interface

These considerations are reflected in our design of the *Persistent Software Transactional Memory* (PSTM) interface, an extension of Haskell's STM implementation. PSTM closely resembles the interface of STM and programmers can simply define what variables inside a transaction should be persisted rather than how to persist them. Listing 3 shows our interface extensions to STM.

```
data PTVar a
getRoot    :: a → IO (PTVar a)

newPTVar   :: a → STM (PTVar a)
readPTVar  :: PTVar a → STM a
writePTVar :: PTVar a → a → STM ()
```

Listing 3. PSTM API extension of STM

We introduce new persistent transactional variables (PTVars) analogous to STM's TVars, but residing in PM. Existing programs can be easily adapted to use PSTM by exchanging TVars with PTVars without changing the semantics of the program. The use of the STM monad also for PTVars enables the composition of joint atomic transactions including volatile TVars and persistent PTVars (KP 1). All modifications of PTVars in a single transaction are performed power-fail safe (KP 3) and the transitive persistence of newly assigned expressions is automatically ensured (invariant 3i).

Expressions are persisted by transitively copying them into PM along with all volatile parts of their environment when the `IO` action `atomically` is called to commit the transaction. As this process is hidden from the programmer, unexpected overheads can emerge during the deep copy of possibly unevaluated expressions to PM. Currently, this is the default trade-off we have chosen in order to preserve the abstraction from the memory layer. For more fine-grained control over structures entering PM, deep evaluation can be optionally enforced by the programmer to prevent performance impacts for expressions that are comprised of large environments.

`readPTVar` establishes a direct reference to an expression in PM (KP 2) which can be passed outside the transaction by using `atomically`. By differentiating between TVars and PTVars at type level, the programmer can be certain whether interacting with persistent or volatile state (KP 6).

```
main = do
    root <- getRoot Nothing
    rval <- atomically $ readPTVar root
    x <- case rval of
        Nothing → atomically $ do
                      pv2 <- newPTVar 5
                      writePTVar root (Just pv2)
                      return pv2
        Just pv2 → return pv2
    y <- atomically $ newPTVar Nothing
```

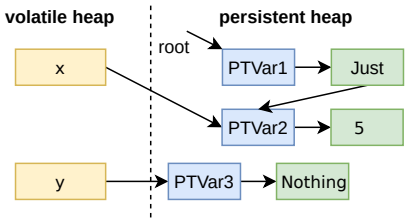


Fig. 2. Creation of structures spanning volatile and persistence domain.

3.3 Identification of Persisted Data across Runs

PSTM must offer a possibility to retrieve variables across runs (KP 4). The PM region is obtained by the application via a file on a filesystem which allows direct mapping and ensures valid access permissions (KP 8). The root PTVar returned by the `getRoot` function provides the necessary anchor to persistent data. It is assumed to act as an always existing reference into the persistent heap while its value might need initialization. The programmer can flexibly define the type of value attached to the root variable. This is done by passing an initial expression to the `getRoot` function that determines the structure of the value, e.g. a list of other PTVars. Once it is assigned by either a previous program run that used the same PM region or the first call of the function, further usages refer to the already set root and the passed initialization parameter is ignored. Our general advise is to use `getRoot` just once during the initialization phase of the program in order to obtain the existing root or initialize a new root otherwise.

The automated memory management reclaims all structures in PM which are no longer reachable from either the volatile heap or the persistent root. That is, all persistent data that should be accessible across program runs must be made reachable from the root (KP 5).

Retrieving data structures across runs is a trivial task for programs that persist a single or only few data structures. In the example in Figure 2, `PTVar1` is returned by `getRoot`. `PTVar2` can be reached from it in consecutive runs. `PTVar3` is also stored in PM and is directly referenced by `y` (KP 2), but it is not available in succeeding runs as it has no connection to `PTVar1`. In theory, persistent expressions that are only referenced from the volatile expressions could be moved back to the

```
getPTVar :: String → IO (Maybe (PTVar Int))
getPTVar key = do root <- getRoot Map.empty
                  atomically $ do
                      map <- readPTVar root
                      return (Map.lookup key map)

insertPTVar:: String → PTVar Int → IO ()
insertPTVar key tv = do root <- getRoot Map.empty
                        atomically $ do
                            map <- readPTVar root
                            let map' = Map.insert key tv map
                            writePTVar root map'
```

Listing 4. Providing named access to PTVars by storing a map in the root variable

volatile heap. However, by keeping them in PM the persistent heap can also function as a volatile heap extension (`PTVar3` in our example). PSTM currently only supports multiple runs of the same binary for implementation reasons (see Section 4.1), thus static type checking is sufficient as no external application can alter the types defined during compile time. In our example, the root PTVar has the static type `Maybe (PTVar Int)` across runs.

As `PTVar2` was recovered directly through `PTVar1` no explicit persistent identifiers were required. However, more complex applications might need additional mapping structures to reload an arbitrary set of PTVars between runs. Complex mappings can be flexibly built on top of the root variable. For example, we can provide named access to PTVars by storing a `Map String (PTVar Int)` in the root variable as shown in Listing 4.

3.4 Persistent Laziness

With PSTM even functions and unevaluated expressions can be assigned to PTVars and thus be stored in PM, e.g. infinite lists. Our design is fully compatible with the lazy execution model of Haskell by providing the corresponding runtime support (KP 7). Persistent expressions are reduced as needed and are directly replaced by their results without any programmer intervention required. This enables the development of applications that are inherently restartable and remember execution state across runs.

Listing 5 shows a simplified time integration implementation based on a particle simulation. In the first run (when the root variable is `[]`), we load the initial particle state, possibly from an external configuration file. We then generate an unevaluated infinite list of steps using `iterate` and store it in the root variable. After initialization, we request the result of the 1000^{th} time step such that all preceding time steps are executed lazily and their results are persisted. If the application is interrupted in between, the evaluated particle configurations of the already performed time steps are kept in the list and the computation resumes from the last executed step seamlessly.

In general, storing unevaluated expressions in PM can trigger multiple copy processes, one for the unevaluated structure and one additional for its result value after evaluation. In some circumstances it can be beneficial for experienced programmers to enforce only strictly evaluated expressions to be stored in PM. However, our data structure benchmarks (see Section 5.2) have shown that unevaluated structures can also have more compact memory representations and thus offload work from the transaction mechanism. Whether strict evaluation is more efficient or not highly

```
doStep :: [Particle] → [Particle]
doStep pl = ...

main = do
    root <- getRoot []
    rootv <- atomically $ readPTVar root
    when (rootv == []) $ do
        initialPState <- loadInitialConfig
        let stepList = iterate doStep initialPState
        atomically $ writePTVar root stepList

    -- retrieve list of steps (of type [[Particle]])
    steps <- atomically $ readPTVar root
    print (steps !! 1000)
```

Listing 5. Example of a persistent and restartable time integration application.

depends on the computational overhead of the lazily evaluated structures compared to the copy overhead, whether they will definitely be used, and on the memory footprint of the unevaluated vs. the evaluated structure.

3.5 PSTM and Orthogonal Persistence

We describe our solution as nearly achieving orthogonal persistence. Due to direct access to persistent expressions and transparent movement of values into PM, no explicit synchronization is required by the programmer and thus *persistence independence* is provided. Moreover, by providing automatic memory management to manage the lifetime based on reachability from either the volatile heap or the persistent root variable, *persistence identification* is guaranteed. However, there exist certain data types that we currently do not support to be persisted, weakening *data type orthogonality*. These are mutable variable types in PM distinct from PTVars, such as MutVars (mutable references), mutable Arrays, or MVars which are variables, used for communication between concurrent threads. These could be integrated by implementing specialised update mechanisms that guarantee power-fail safe consistency. For MVars that are used for blocking of threads albeit the question arises how reasonable it would be to persist them. However, they could be released by the recovery mechanism to prevent them from blocking across runs. We focused this work on transactional variables which are more universal and offer higher consistency guarantees. We have chosen a rather explicit approach to persistence by differentiating between TVars and PTVars. In principal, they could be expressed by the same type and truly orthogonal persistence would be achieved by maintaining their persistence in correspondence to their reachability from the root variable and move them to PM transparently.

4 TECHNICAL REALIZATION

The runtime system (RTS) of Haskell controls the execution of Haskell programs, provides transaction support, and performs automatic memory management. The Haskell RTS did not include any support for PM before the introduction of our PSTM extensions. We have therefore extended the STM implementation to support persistent transactions and GHC's memory management by an additional heap that resides in PM. To fully support Haskell's call-by-need semantics, we allow the lazy evaluation of persistent expressions in PM. Finally, a new recovery step ensures consistent PM state across program runs.

4.1 Persistent Heap and Automatic Memory Management

We introduce a hybrid heap design which extends the volatile heap with a persistent region. It resides in a file passed as a runtime parameter (or created in the first run) which is directly mapped into the virtual address space and can be accessed across runs. At program startup, the volatile and the persistent heap are mapped to fixed, subsequent address ranges. The fixed mapping ensures that references in PM retain their validity across runs and eliminates the need for offset-based pointer resolution at runtime.

A Haskell program is represented as a graph in volatile memory. Most closures, i.e. constructors and functions, in this graph are immutable. Only variables and unevaluated expressions, i.e. thunks, are mutated in-place. Assignments in transactions trigger the automatic copy process of these expressions to PM and allow to replicate parts of the volatile graph in PM. All memory objects in PM are stored in their native closure layout and can be directly accessed, so that data does not have to be serialized before it is copied to PM. This native closure layout contains pointers to code in the statically-linked binary, which currently restricts PM structures to be used by a single application [Berthold 2010; Yang et al. 2015]. To support multiple binaries the code needs to be shared between them, e.g. through dynamic linking, and the compatibility of Haskell data types across binaries

must be ensured as they are not stable across recompilation. One solution could be the approach of the *Unison* language [Chiusano 2020], which leverages hash based identifiers for definitions and data declarations. These are stable across renaming and recompilation and allow sharing of the code base between applications. When resolving these issues, distinct binaries that act on the same PM region could be supported by including dynamic type-checkers [Abadi et al. 1991; Cheney and Hinze 2002; Connor 1991]. The transaction mechanism and memory management on the same region nevertheless would need to be externalized if concurrent usage is intended.

Haskell programs constantly create new subgraphs of closures while others become unreachable. We provide automatic memory management for closures residing in PM to ensure that the user does not have to explicitly manage their lifetime. The Haskell RTS uses a stop-the-world generational copying garbage collector for its volatile heap as default. Repetitive copying of data structures in PM, which possibly outlive multiple runs, could induce a large overhead. We adapted the recently introduced nonmoving *Alligator* collector [Gamari and Dietz 2020] to mitigate these overheads. The nonmoving collector is integrated with the generational copying collector and operates on the memory region containing the oldest generation. A separate thread is used to concurrently garbage collect a snapshot taken during the stop-the-world phase of the generational copying collector. The snapshot state contains the collection roots, i.e. the closures in the nonmoving region that were found to be reachable by younger generations.

As a proof-of-concept, we integrated the PM heap as part of this oldest generation. A closure in PM is alive if it is reachable from either the persistent root **or** structures in the volatile heap. Garbage collection of the PM heap is triggered with the regular collection cycles. However, the nonmoving collector currently does not limit the heap size, whereas the size of the PM file is limited at runtime to prevent it from growing indefinitely and persistently occupy large amounts of PM. We introduced an additional triggering mechanism when running out of persistent heap space. Haskell threads that require PM allocation are then suspended until the triggered nonmoving garbage collection finishes. In future versions, they could cooperate with the collector thread to speed up the collection and reduce the resulting mutator pause times.

We introduce a new PM heap allocator extending the design of *Alligator's* class-based allocator which serves allocation requests in the nonmoving heap. The PM heap is structured in correspondence to the layout assumed by the RTS, i.e. block descriptors at the beginning of each megablock boundary are provided. It is further divided into segments that provide sub-blocks used to allocate individual closures. Each segment has a state that describes its occupancy. Our allocator resides in PM and the allocator state needs to outlive one run. Internally, states are represented as linked lists, e.g., the global list of free segments, a thread-local list of partially filled segments or the list of segments currently under collection. To prevent memory space leaks while moving segments, we use a redo log in PM that stores state changes so that they can be recovered in the event of a crash. The liveness of closures within a segment is described by a bitmap in the segment's metadata and updated during collection. Keeping those liveness bitmaps of the segments power fail-safe consistent would result in large logging and flushing overheads in the collectors mark phase. Instead, we decide to restore valid bitmap states during an initial heap traversal during recovery (see 4.5).

4.2 Persistent Transaction Support

Figure 3 shows the execution of a PSTM transaction inside the RTS. Figure 3a shows the initial memory state. The thread-local execution of the transaction creates a log that captures the read and write set of the transaction using references to the respective closure graphs in memory, shown in Figure 3b. In these phases the native STM mechanism is leveraged and there is no difference for PSTM, except for the possibility to create new PTVars in PM.

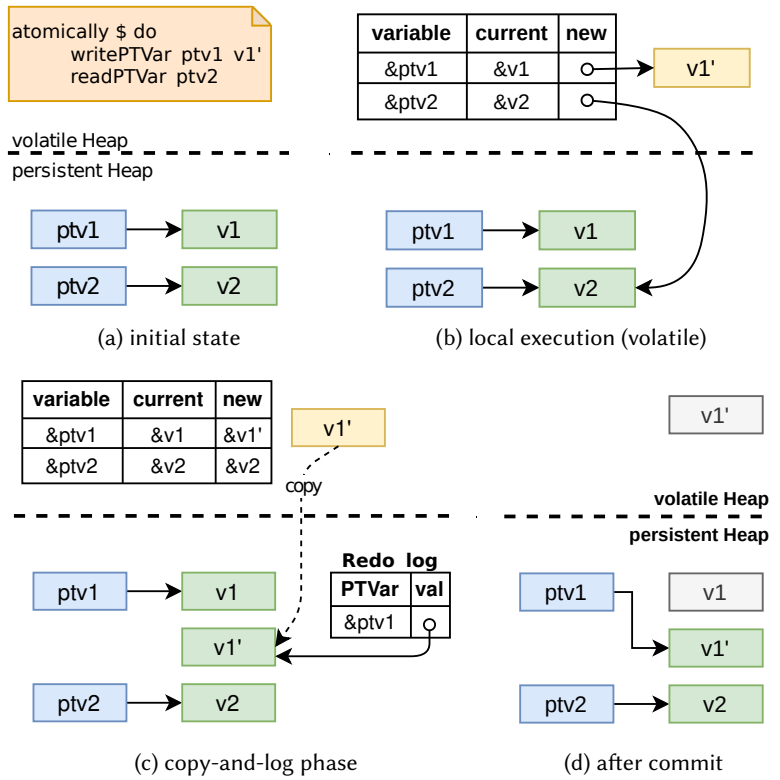


Fig. 3. Example PSTM transaction: changes are prepared locally in the volatile heap in **(a)** and **(b)**. In **(c)** the valid transaction commits. Values are moved to PM and changes are redo-logged. In **(d)**, the previous values (grey) can be reclaimed by the automatic memory management.

A transaction enters the validation phase after its thread local execution (step between Figures 3b and 3c). The validation step locks all involved variables and compares them to the log's read set to ensure that these variables have not been modified by concurrent transactions. STM locks a TVar by overwriting its global value field with a pointer to the thread local log containing its actual value. If the validation fails, all involved TVars are unlocked by recovering the value from the log and the thread local execution is restarted. We do not persist the transaction log; thus pointing from the value field of a PTVar to it would result in data losses between runs and the actual value cannot be recovered. We have introduced a separate lock field inside PTVars that ensures that the global value is never overwritten to allow recovery after power failures. To choose the appropriate locking scheme TVars and PTVars are distinguished based on their memory location.

A transaction may commit if the validation succeeds. We have extended the commit logic of STM by a copy-and-log phase for all involved PTVars to ensure that they always reference values in PM (Invariant 3ii). We transitively copy all reachable volatile closures to PM if a newly assigned value resides in the volatile heap (Figure 3c). The copy process can be truncated every time it reaches closures which already reside in PM as these closures may only point to other persistent closures. Postponing the copy process to the commit phase avoids unnecessary copies of thread-local state due to re-execution. However, some closures might be currently under evaluation by another thread and might be blocked by a blackhole. These cannot be copied power-fail safe in the current RTS

implementation, as they neither contain a reference to the original thunk computation to revert them, nor represent a valid result and thus have no meaning across runs. Instead, we restart the transaction if a blackhole is encountered and block on it until the pending computation is finished. Similarly, the transaction is restarted when the copy process fails for allocation reasons and a garbage collection is required.

The following four steps need to be performed to update PTVars in a power-fail safe manner (Invariant 3i):

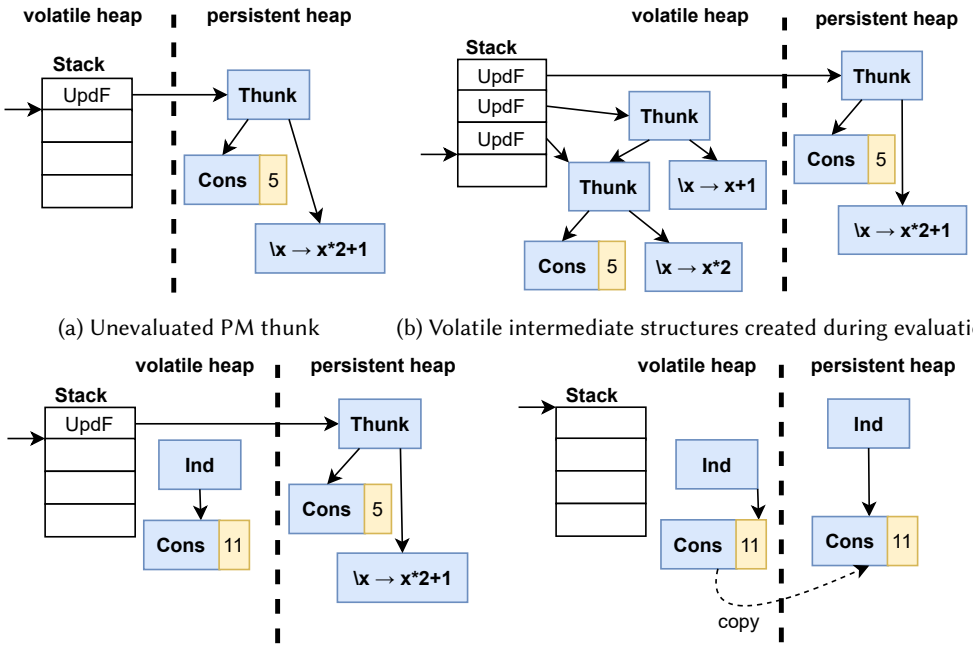
(1) Copy newly assigned values of all involved PTVars transitively into PM and record the mapping of modified PTVars to new PM replicas in a redo-log
(2) Activate the redo-log; from this point all PTVars are guaranteed to receive the update
(3) Update all modified PTVars to their new value
(4) Invalidate the redo-log

Before each step, all modifications of the previous step need to reach the persistence domain. We use libpmem's (which is part of Intel PMDK) optimized operations to asynchronously flush all changes performed by the prior step from the CPU cache to PM. This is followed by a memory fence to ensure a consistent ordering with writes of succeeding steps. Note that we only log updates of PTVars, but do not require to control updates of their values as they are immutable, except for unevaluated computations that are controlled separately. The commit phase is completed by unlocking all TVars and PTVars. As we use redo-logging of the write-set we can unlock read-only variables directly after the validation phase without violating transaction ordering as an optimization to improve concurrency. Figure 3d shows the memory state after the transaction. Unreachable structures in the volatile or persistent heap, including those created by transaction restarts, will be reclaimed by the automatic memory management.

Concurrency-safety is subject to the original STM mechanism and controlled per transactional variable identified by its memory address. PTVars are created directly in PM and their memory addresses do not change (e.g. by moving them from the volatile heap to PM). Moreover, TVars as well as PTVars are locked during validation and commit phases. Flushing PTVar lock fields is not required (in fact, they could also be volatile), as a consistent view on them is ensured by cache coherence at runtime and they are reverted on crash recovery. Thus, the concurrency control is not impaired. PTVars are not locked by overwriting the global value field. This does not impair atomicity as the validation of local execution is only performed when all locks are held, thus the read set is compared with an atomic view of the global state. Regarding power fail-safe atomicity, Step 1 does not change any variable directly, while Step 2 activates the log in an atomic operation and contains all information to reflect the updates in case of power failures and the transaction thus is guaranteed to commit. Although the new values are not necessarily reachable for automatic memory management as they are not integrated into the closure graph, no snapshot can be taken for the garbage collector while a transaction commits and thus the new values survive until successful commit or potential recovery. If Step 3 is interrupted, reassignment of any value during recovery does not collide with other changes, as logically PTVars were not released and no other transaction was allowed to modify them before the log was disabled in Step 4. Thus updates by PSTM are durably consistent and the ACID guarantees are fulfilled.

4.3 Unevaluated Expressions in PM

Unevaluated Haskell expressions are represented as thunks. Figure 4 shows the evaluation of thunks in PM which are performed orthogonal to the transaction mechanism. The entry code of a thunk defines how the result value should be computed. For each thunk that needs to be evaluated, a special stack frame, called *update frame*, is pushed onto the stack (Figure 4a). It is responsible for



(a) Unevaluated PM thunk (b) Volatile intermediate structures created during evaluation

(c) Intermediate thunks were updated to result (d) The PM thunk was updated by its update frame

Fig. 4. Example of persistent thunk evaluation: when a thunk is evaluated, an update frame is pushed **(a)** and the thunk's entry code is entered. This might create short-lived intermediate structures **(b)**, which are reduced subsequently **(c)**. Finally, the PM thunk's update frame is again the top-most stack frame and the evaluated result is copied and integrated into the graph **(d)**.

integrating the evaluation result into the closure graph and updating the thunk with an indirection to its result value.

We have extended the logic of *update frames* to support lazy evaluation for persistent thunks. PM thunks can create many very short lived intermediate thunks which represent sub-expressions as illustrated in Figure 4b. We store results of intermediate thunks (Figure 4c) in fast DRAM and only the final results are copied (and flushed) transitively into PM (Invariant 3ii). Thunks in PM can then be updated with an indirection (Figure 4d) through a single and power-safe pointer update. Our automatic memory management supports to shortcut these indirections using atomic pointer updates allowing them to be reclaimed.

Thunks describe pure function applications and they can be evaluated multiple times by concurrent threads without changing their result. However, thunk headers can be replaced by a blackhole header to reduce compute overheads. This happens outside of the control of the update frame. The original thunk header would, without further protection, be lost if a power failure occurs while it is blackholed. We therefore store the original thunk info pointer as undo-information in front of the closure when copying the thunk to PM. Thus, blackholes can be reverted to thunks in succeeding runs.

4.4 Sharing of Structures in PM

In Haskell, the duplicate creation of the same (sub-)expression can be avoided by sharing heap structures. Top-level definitions are shared globally while definitions through `let` are shared in

their appropriate scope. If a shared expression is a thunk, it is evaluated once and all subsequent usages refer to the result. PSTM often implicitly copies volatile expressions to PM by assigning them to PTVars. This can lead to multiple evaluations of thunks as a volatile as well as a persistent version exists. However, Haskell's purity ensures that this only creates additional work and all evaluations lead to the same result.

We preserve sharing within a given PSTM transaction, i.e. volatile sub-expressions reachable from multiple PTVars in the same transaction are only copied once. Internally, we use a map that tracks the location of all newly copied values. This also allows us to copy cyclic structures as we can trace whether a closure was already copied. We do not preserve sharing across multiple transactions to avoid maintaining a global mapping of all closures and their PM replicas. Efficiently sustaining such a mapping across runs would indeed be challenging as closures can be dynamically allocated and are identified by their memory addresses, which are not unique between runs. Here, proper indexing mechanisms to compare and find closure (sub-)graphs that contain the same expressions are required. Assigning the same volatile value in distinct transactions therefore creates multiple copies of this value. The programmer can prevent this overhead by assigning a volatile value to a PTVar in a separate transaction to retrieve a direct reference before assigning its, now persisted, value to other PTVars in following transactions.

Care needs to be taken for top-level definitions which are allocated statically and their memory locations are thus known at compile-time. Resulting *static* closures are created as part of the binary in a section called *static region*. In contrast to references between heap closures, they are not necessarily referenced via other closures' payloads as an optimization of the compiler. Instead, the entry code of a closure can hold direct memory references to these static closures [Ağacan 2020]. We copy a heap closure as well as all closures referenced from it to PM and reflect the previous relation by updating the payload pointers. However, entry code references cannot be easily updated on a per-closure basis as done for payload pointers as the same entry code might be shared between multiple closures. This is illustrated in Figure 5, where a volatile heap structure as well as its PM replica is shown. Both still point to the same entry code and consequently to the same static closure. This is not an issue for immutable static closures such as constructors and functions as they are available in every run. However, also mutable static thunks, called *constant applicative forms* (CAFs), exist. CAFs are normally replaced by indirections into the volatile heap during their evaluation. Already evaluated results of CAFs are therefore lost in succeeding runs when the static region is reloaded from the binary. We copy CAFs or their results which are reachable from a persistent closure to PM and update the original CAF to an indirection to the new PM CAF (illustrated by

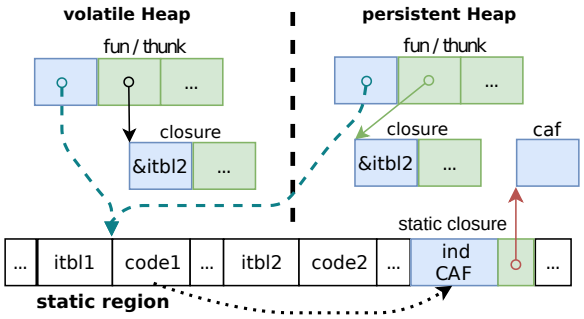


Fig. 5. CAFs are copied to PM if reachable from a PTVar and replaced with a persistent indirection (red arrow).

the red arrow). The mapping between the original CAF and the PM CAF is held persistently to allow its recovery across runs. Moving CAFs to PM ensures that effectively only one version, either volatile or persistent, exists and thus the sharing of static closures is preserved.

4.5 Recovery Mechanism

We perform recovery during runtime system initialization. PSTM logs can be applied in any order to redo uncommitted changes as each PTVar is at most contained in one log. Note that there is no need to revert interrupted copy processes. Structures are either copied completely before they are made reachable in the live PM graph or are automatically garbage collected. If the program crashes while a segment state is changed by our memory management, i.e. while it is moved from one state list to another, the move operation is completed after the crash.

Since the locks of PTVars are part of the PTVar structure, they must be reverted to a valid state which is done in a traversal of the PM heap starting from the root. Here, also interrupted computations which are still blocked by blackholes are reverted to the original thunk. Additionally, all CAFs that are reachable from the persistent root are restored. This process has an overhead comparable to the mark phase of the garbage collector as each closure is traversed once. In fact, the process is also used to set the segments liveness bitmaps as we do not rely on flushing them to PM consistently. Running the recovery mechanism could be often avoided as an optimization by identifying graceful application shut-downs. This would require to track the CAFs reachable from PM and flush the valid segment bitmaps before shutdown.

5 EVALUATION

In this section, we first compare PSTM with publicly available library approaches that offer persistent transactions and analyze the overhead of garbage collecting PM structures. Next, we empirically measure the overhead of persistent compared to volatile STM transactions. Additionally, the cost of persistent lazy evaluation is measured based on persistent memoization. We provide our implementation of PSTM, all benchmarks as well as pre-built artifacts as open source[1]

5.1 Methodology

The performance measurements have been conducted on a single-socket server comprised of a 10-core, 20-thread Intel Xeon Gold 5215 processor with 192GB DRAM and 6x128GB Intel Optane DC persistent memory running CentOS 8. PSTM has been integrated into the GHC 8.9 branch *nonmoving-compact* [Gamari and Ağacan 2019]. It provides an implementation of the volatile nonmoving garbage collection strategy described by Gamari and Dietz [2020]. Benchmarks for STM were run using a slightly adapted version of the more recent GHC 8.10.1 that allows capturing STM commit statistics with an activated nonmoving collector.

As we want to ensure best comparability, our evaluation is based on data structures and transaction mechanisms that are peer-reviewed open source projects. The Haskell benchmarks are based on the benchmark suite for STM by Yates and Scott [2017, 2019], which we adapted to also work with PSTM. The suite provides concurrent tree set implementations, including a red-black tree (RBTREE), a randomized binary search tree (TREAP), and a hash array mapped trie (HAMT). Additionally, we modified an existing concurrent hash table (HT) Haskell library [Robinson 2019] to use PSTM instead of STM.

- **Red-Black Tree**. Red-Black trees are self-balancing binary search trees that allow for lookup, insert and delete operations in $O(\log n)$. By labeling each node with either *black* or *red* it can be ensured that the longest path in the tree is at most twice as long as the shortest path.

[1]Repository can be found at https://gitlab.rlp.net/zdvresearch/haskellpstm

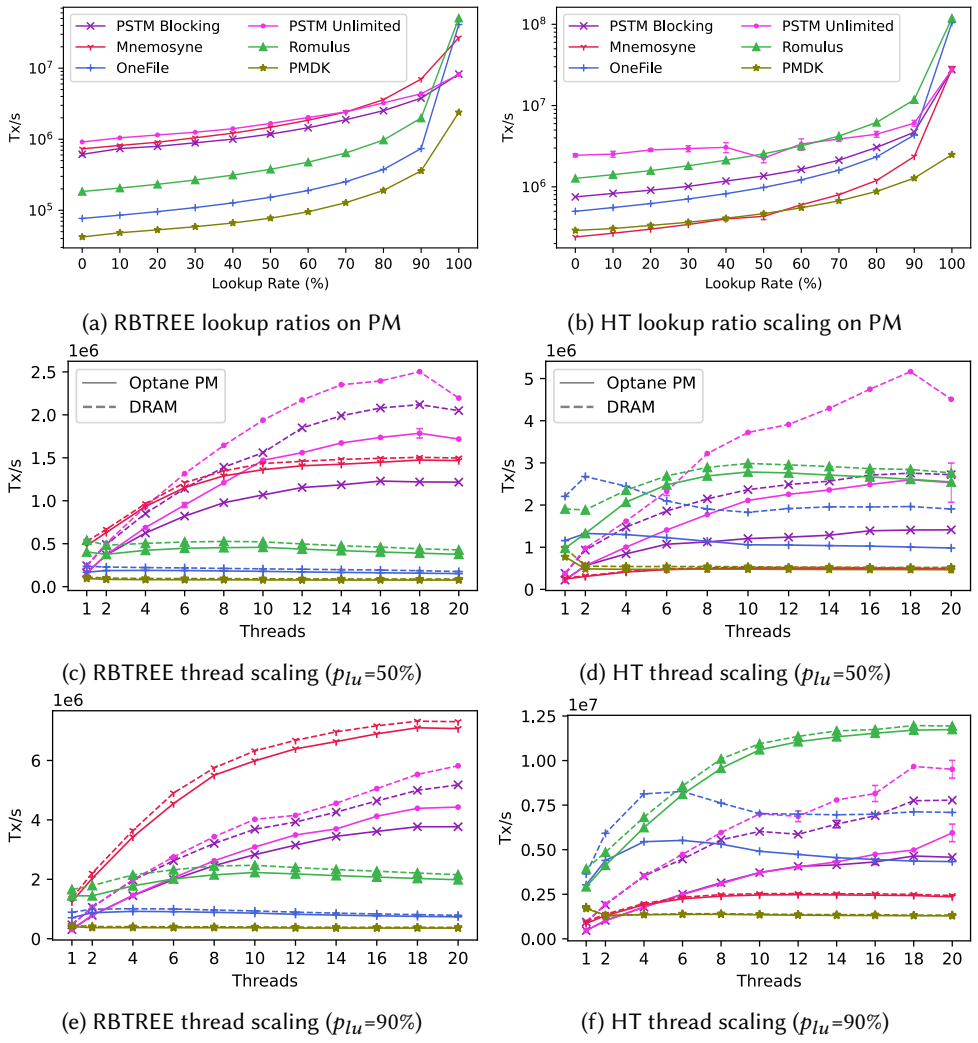
Rebalancing is performed during insert and delete operations ensuring that no two adjacent nodes are both red and that each path from root to leaf contains the same number of black nodes. In addition to the color, each node contains a key-value pair as well as pointers to the parent and the two child nodes.

- **Treap**. A treap is a combined binary tree and heap data structure. A node in the tree consists out of two (P)TVars modelling the modifiable relation to the child nodes, as well as a key-value pair and a priority. While the key is used to maintain the left-to-right ordering in the tree, the priorities are assigned randomly and define at which depth a node has to be placed. Insertion of nodes at random heights leads to rotations that are expected to maintain logarithmic height, independent of the insertion order. Since the priority defining the height is obtained randomly and each lookup of a key starts from the root, it is likely that a transaction inserting a node nearby the root invalidates the read sets of concurrent transactions that also rely on that path, thus leading to conflicts.
- **Hashed Array Mapped Trie**. A Hashed Array Mapped Trie (HAMT) is the combination of array mapped tries and hash tables. Each key is first hashed. The (hashed) key-value pair is then inserted into a trie [De La Briandais 1959; Mäsker et al. 2019]. The trie has two different kinds of nodes – levels and leaves. A level represents an n-bit chunk of a key. Leaves contain key-value pairs. The hashing of the key ensures a well-balanced trie which tends to get more sparse towards the leaves. For a given maximum key size the trie has a constant depth. By mapping it onto an array this sparsity can be leveraged to allow for more economical usage of available memory.
- **Hash Table**. A Hash Table uses a hash function to map keys directly to values. Given a key and the hash function an index is computed which points into an array of buckets. Buckets are implemented as a mutable linked list. Inserts and deletes only create writes in the corresponding bucket which keeps transaction logs small. For supporting arbitrary numbers of key-value pairs dynamic resizing of the array is desirable, as it keeps the mean size of the linked lists small. However, this induces large copy overheads.

The data structures have been initialized with 50,000 elements out of a key space of 100,000 elements to ensure comparability with the benchmarks of Yates and Scott [2019]. Worker threads perform lookup, insert, and delete operations. The lookup rate (p_{lu}) defines the ratios of the respective operations. Inserts and deletes are performed with equal probability, given as $(1 - p_{lu})/2$. Thus, the number of elements in the structure is expected to remain constant. All benchmarks were averaged over 5 runs. Each run has been restricted to a runtime of 60 seconds, except for Mnemosyne where we were restricted to 3 seconds as we observed an unstable behavior for longer runtimes. The relative standard error of the mean (SEM) was (far) below 3% for most of our results and we include error bars for larger deviations. The experiments also include results for PSTM and other persistent libraries completely running on DRAM to understand the impact of faster PM technologies coming up in the future. PSTM and the other libraries running on DRAM still perform all operations according to the standard PSTM protocol, whereas of course the data on DRAM cannot be recovered in case of a crash.

5.2 Comparison with Library Based Approaches

We compare our approach with the libraries OneFile [Ramalhete et al. 2019], Romulus [Correia et al. 2018], PMDK [Rudoff 2017], and Mnemosyne [Volos et al. 2011]. We used a benchmark suite by Ramalhete et al. [2019] that offers a red-black tree and a hash table implementation for OneFile, PMDK and Romulus. It ensures concurrency-safety of PMDK transactions by using a global lock. While specialized implementations can leverage more fine-grained synchronization approaches,



(a) RBTREE lookup ratios on PM

(b) HT lookup ratio scaling on PM

(c) RBTREE thread scaling (p_{lu}=50%)

(d) HT thread scaling (p_{lu}=50%)

(e) RBTREE thread scaling (p_{lu}=90%)

(f) HT thread scaling (p_{lu}=90%)

Fig. 6. Red-black tree (left) set and hash table (right) transaction throughput (Tx/s) scaling behavior.

the abstraction of the global locking approach is comparable to STM. For Mnemosyne we used the red-black tree and hash table implementations of the vacation example of STAMP [Minh et al. 2008; Nalli 2018]. We have not been able to compare PSTM with Java-based implementations, as AutoPersist and Espresso have not been available as publicly available repositories. We adapted all benchmarks to the workload used by Yates and Scott [2019] described in the previous section. While the implementations differ across languages, the well-defined operations on red-black trees and hash tables ensure the same workload. As Mnemosyne has internal limits on the transaction read and write set size, we used a fixed number of 6000 buckets for all hash table benchmarks. OneFile provides two concurrency approaches called `OneFilePTM-LF` and `OneFilePTM-WF`. Both show very similar results for these benchmarks and we therefore only present the results of `OneFilePTM-LF` as OneFile. The same applies to `RomulusLog` and `RomulusLR` where we only show `RomulusLR` as Romulus. The library approaches reclaim memory objects explicitly when they are replaced. To

allow a fair comparison, we thus limit the heap size to 1500 Mb for PSTM to enforce a timely collection of unreachable structures in PM. The mutator threads are *blocked* if the mutator outruns the collector until space was reclaimed. However, we also include an *unlimited* configuration with a large PM heap file where garbage is only collected at regular collection cycles. Here, the heap is allowed to grow over time to show PSTM's scaling potential when lower garbage collection overheads occur.

Figures 6 (a) and (b) show the impact of different lookup ratios on the transaction throughput (Tx/s) for 20 concurrent threads for RBTREE and HT. These graphs only show results created on Optane DC PM, as the DRAM results have shown very similar trends on this log-scale projection. PSTM shows comparable performance to the library solutions and outperforms many approaches for lookup ratios lower than 80%. Due to the optimistic synchronization approach of (P)STM by concurrently preparing transactions in local state, a higher degree of parallelism at high update rates can be achieved. The throughput of the other approaches decreases more drastically when many write transactions are involved. Moreover, both shown data structures leverage lazy evaluation for PSTM which allows to offload both, copy and computation overhead, from the transaction mechanism. For RBTREE, we measured that approximately $1.2x$ more bytes were copied to PM during lazy thunk updates than during the transaction commit phases, while thunk updates were responsible for 2/3 of the bytes written to PM for HT. Updates of the RBTREE can cause rebalancing steps and therefore many write operations per transaction. Only Mnemosyne shows a similar performance compared to PSTM for the RBTREE when writes are involved, as it relies on a similar redo logging mechanism. However, Mnemosyne needs to log all changes as well as allocation as it does not integrate with an automatic memory management that prevents memory space leaks. PSTM only has to use a redo log to allow multiple power-fail safe pointer updates of PTVars. Therefore *PSTM unlimited* shows that PSTM can outperform Mnemosyne for write intensive workloads due to more light-weight transaction logging when garbage collection impacts are reduced. Romulus does not allow concurrent writes in transactions and the parallelism therefore degrades. OneFile allows to cooperatively process the write set of one transaction with multiple threads but this also induces a communication overhead. However, when only few variables are involved and writes are sufficiently small, as for HT, OneFile and Romulus can outperform Mnemosyne. Although PMDK transactions are executed sequentially, the used undo logging mechanism which needs to capture the state prior to modification also introduces severe overheads on write transactions.

Figure 6 (c) and (e) show the thread scaling behavior of the red-black trees for 50% and 90% lookup rates. Generally, only Mnemosyne and PSTM show significant benefits from parallelism. At 50% lookup rate, PSTM scales well for low thread counts while the throughput improvement slows down for higher thread counts. For thread counts above 10 hyper-threading is used. Also, the garbage collection impact increases noticeable with the number of threads for the blocking version as will be shown in more detail in the next section. In contrast to the library approaches, PSTM's throughput diverges more between DRAM and Optane DC the more threads are involved. This is due to bandwidth limitations of Optane DC as its maximum write bandwidth can only be maintained for writes of 256 bytes and above [Izraelevitz et al. 2019]. Future work could further optimize the allocator to match optimal write characteristics of the underlying PM hardware. For 90% lookup ratio this divergence is decreased, as less write transactions are involved. The impact of garbage collection for PSTM becomes visible for 20 concurrent worker threads. The garbage collection is performed by a dedicated OS thread and the throughput of the worker threads is therefore impacted when the maximum hardware thread count is reached.

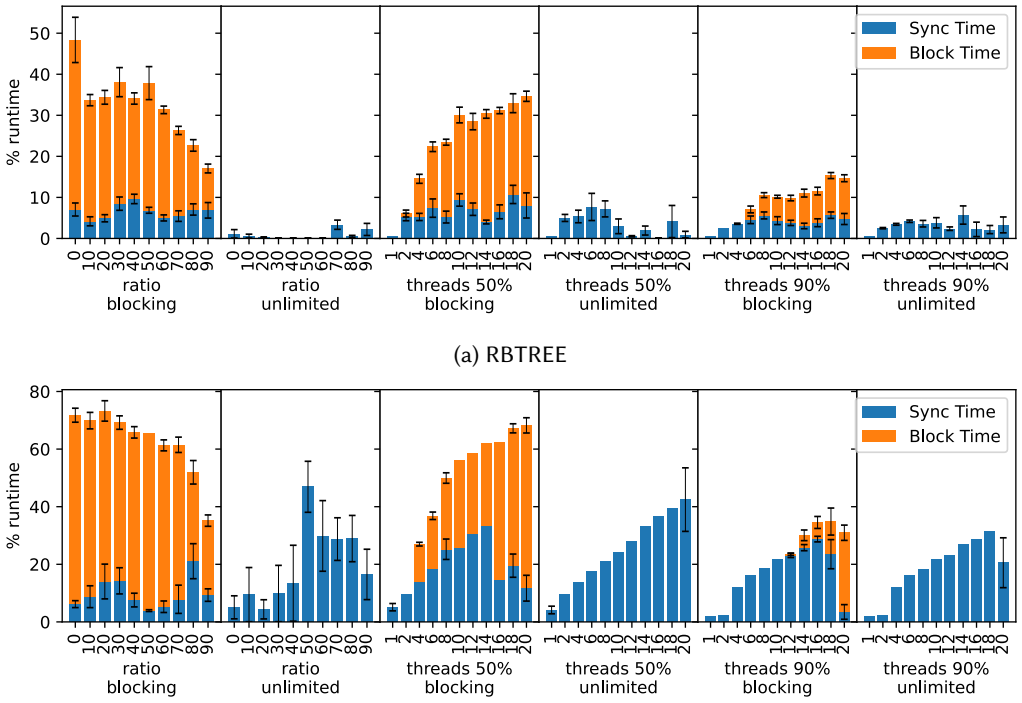
Figures 6 (d) and 6(f) show the respective thread scaling results of the hash table. For a lookup rate of 50% it is noticeable that the performance of Mnemosyne is in the same order of magnitude as sequentialized PMDK transactions. Here, the benefit from parallelism is lower as the commit

phase of the transaction mechanism imposes an overhead for few writes while conflicts are unlikely. PSTM's overhead scales with the number of variables that participate in the transaction. OneFile and Romulus scale with the likelihood of concurrent write transactions, which increases if more threads participate.

5.3 Garbage Collection Impact

The previous results show large variance in throughput for HT for PSTM induced by blocking times of the garbage collector. The nonmoving collector runs concurrently to the mutator which can modify mutable memory objects, such as PTVars, while the collector marks the reachable closures. In this phase mutator modifications are tracked in a so-called *update remembered set* to remember the existing references at snapshot time. They are marked in a subsequent stop-the-world phase after completion of the concurrent mark phase, resulting in *sync times* [Gamari and Dietz 2020]. When the maximum heap size is reached, as in the *blocking* runs, an additional collection is triggered and threads requesting allocation in PM are blocked until space has been reclaimed, resulting in *block times*.

Figure 7 reports the sync and block times for the RBTREE and HT benchmarks as a fraction of the runtime for the Optane DC runs. The sync times for the RBTREE *unlimited* benchmark are relatively low, while they are more severe for HT *unlimited*. The sync time is largely dependent on how many closures in the *update remembered set* have been missed during the concurrent



(a) RBTREE

(b) HT

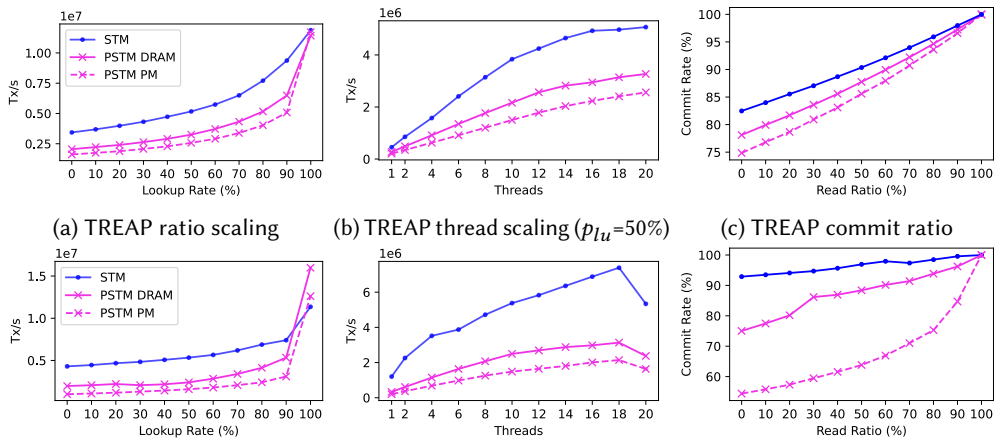
Fig. 7. Blocking times of the nonmoving garbage collector for the RBTREE (top) and HT (bottom) benchmark with and without extra blocking to enforce a limited PM heap size.

mark phase and thus need to be marked retrospectively. For RBTREE many updates of PTVars are caused by rebalancing where most values are still reachable afterwards and thus already have been traced. Contrary, updating transactions of HT are shorter as less variables are involved and values that were reachable at snapshot time can become unreachable more quickly. Moreover, the higher number of thunk updates leads to more thunk references that were missed and must be traced in the sync phase. It can be observed that the highest sync times for HT *unlimited* occur at a lookup ratio of 50%, where they take almost half of the run time. This is due to the high number of thunks newly written to PM which are then forced, i.e. executed, by the many concurrent read transactions. However, also high deviations in sync times are induced by fluctuating numbers of garbage collection cycles. As shown by the thread scaling results the sync time overhead increases linear with the number of threads, except for 20 threads where the maximum hardware thread count is reached. In the *blocking* configurations the overall blocking times are more predictable, as they scale with both, the thread count and the lookup ratio. The mutator outruns the garbage collector even for low thread counts in the thread scaling examples as the collection of the nonmoving heap is performed single-threaded. An exception is HT at the lookup ratio of 90% where the amount of allocations is low enough that no additional blocking is required for up to 10 threads despite the high rate of missed closures during concurrent marking. As the blocking avoids the update of references by the mutator, it reduces the sync times as seen for HT. The higher sync times for RBTREE were induced by an increased number of enforced collections. In total, mutator pause times of up to 48% of the runtime were observed for RBTREE and 73% for HT respectively.

The nonmoving collector was not designed for maintaining many short-lived structures. However, we see the potential to significantly reduce mutator blocking times in future versions by allowing suspended threads to cooperate in the collection.

5.4 Persistence Overhead vs. STM

In this section we compare PSTM transactions to volatile STM transactions to estimate the overheads of the power-fail safe update mechanism and migration of values to PM. We evaluated two different data structures, a TREAP and a HAMT (Figure 8). We compared STM with PSTM in the *unlimited* configuration where we do not introduce additional blocking as the nonmoving collector of the



(a) TREAP ratio scaling (b) TREAP thread scaling (p_{lu}=50%) (c) TREAP commit ratio

(d) HAMT ratio scaling (e) HAMT thread scaling (p_{lu}=50%) (f) HAMT commit ratio

Fig. 8. Comparison of HAMT and TREAP for STM and PSTM.

GHC version used for the volatile STM benchmarks does not allow to limit the heap size. We observed that for STM benchmarks the mutator can outperform the garbage collector resulting in heap usages of up to 40 GB.

While the TREAP uses strict evaluation, the HAMT leverages laziness. PSTM and STM show similar scaling behaviors for varying read ratios and numbers of threads. Across the different read ratios PSTM showed slowdowns between 1.4x and 1.7x on the same memory technology (1.8x and 2.1x on PM) for TREAP at lookup ratios lower than 90%. For 100% read ratio all transaction mechanisms show very similar throughput. Slowdowns between 1.4x and 2.4x on DRAM (2.4x and 4.3x on PM) can be observed for HAMT using PSTM. Here, PSTM slightly outperforms STM when only read transactions are executed. For STM, the data structure gradually ages, i.e. is moved iteratively to older generations by the generational garbage collector. This results in repetitive copy overheads until it is finally copied to the nonmoving heap, whereas PSTM moves the data structure to the nonmoving PM heap on initialization. The measured overheads of PSTM when writes are involved are partly induced by the need of moving values from the volatile heap to PM in an on-commit copy-phase and the required flushing. Additionally, a write-ahead redo-log is created in PM holding references to the new values of PTVars.

PSTM induces a higher overhead on HAMT than on TREAP, as HAMT works on immutable arrays rather than on single values. It therefore copies more bytes on updates. Averaged across lookup ratios, HAMT copies 107 bytes for each write transaction, while TREAP only copies 44 bytes. Copying more bytes prolongs the commit phase and thereby increases the probability of conflicts between transactions, inducing additional overheads. Figure 8(c) and (f) show the respective successful commit rates. TREAP generally shows a higher conflict rate than HAMT, because an update that triggers a rebalancing of the tree close to the root can easily invalidate all concurrent transactions operating on the same branch. However, for TREAP the overheads of PSTM on the commit rate is negligible, as most updating transactions only operate on nodes of one specific branch not limiting concurrent transactions on other branches. The successful commit rate of TREAP shows a maximum decrease of 4.4% on DRAM (7.6% on PM) for PSTM in comparison with STM. For HAMT, the prolonged commit phase has a higher impact on the number of conflicts. HAMT's compact representation leads to less independent paths for concurrent transactions. A node is an array of other nodes that is rewritten when a new node should be added. This invalidates all transactions involving any paths that share this node. This array is generated in the volatile heap for STM and a single pointer update is sufficient, while PSTM needs to replicate the arrays at commit time, blocking a whole set of paths for a longer time. The conflict rate therefore increases by 18% (39% on PM) for HAMT when using PSTM.

We additionally evaluated the overheads of PSTM to STM for RBTREE and HT. PSTM has shown slowdowns between 1.3x and 1.9x on DRAM (1.5x and 2.7x on PM) for RBTREE and 1.5x and 2.8x on DRAM (2.2x and 4.4x on PM) for HT, respectively. The scaling behavior of PSTM resembled that of STM.

5.5 Persistent Memoization

Haskell's laziness is used in many libraries and is also supported by PSTM. A possible application of PSTM is persistent lazy memoization. Memoization provides a trade-off between computation time and memory consumption of a program. Previously computed results are stored and returned for the same input argument to a function instead of recomputing it. Memoization in Haskell is most commonly implemented using `MemoTrie` [Elliot 2008] and `MemoCombinators` [Palmer 2013]. MemoTrie uses a trie to store previous results. Depending on the function type, MemoCombinators can select from multiple data structures also including a trie for memoization.

```
fib :: (Int → Integer) → Int → Integer
fib f 0 = 0
fib f 1 = 1
fib f n = f (n − 1) + f (n − 2)

memofib :: IO (Int → Integer)
memofib = do tv <− getRoot $ fix (memo . fib)
             f <− atomically $ readPTVar tv
```

Listing 6. Example of persistent lazy memoization.

Listing 6 shows a memoization example using MemoTrie. The higher order function *memo* takes a function and returns the memoizing equivalent. By assigning the memoizing function to a persistent variable, the function as well as its data structure holding the memoized results are persisted. *Fibonacci* heavily benefits from the reuse of previously computed results as each number recursively depends on all smaller Fibonacci numbers. Here, the *fix* function is used to additionally benefit from previously computed values in the same call.

Table 1 shows the wall-clock time required for computing the 150,000th Fibonacci number using MemoTrie and MemoCombinators. All recursive calls map to the same memoized function, thus all computed Fibonacci numbers are shared in the same call to the function. As the libraries heavily rely on lazy evaluation, the measurements serve as an estimation for the cost of thunk evaluation in PM. Persistent memoization shows slowdowns of 2.9x (4.5x on PM) for MemoTrie and 3.5x (4.8x on PM) for MemoCombinators when compared to the volatile counterpart. This cost amortizes quickly when the memoized values are reused across a few runs instead of recomputing them.

The high number of average bytes copied per thunk update is a result of the large Fibonacci numbers (*fib(150,000)* has 31,348 decimal digits). The higher runtime of MemoCombinators is explained by the higher number of individual thunk updates, each requiring the evacuation of results from the volatile to the PM heap. Individual updates are costly because they need flushing and memory fencing. While 150,000 thunk updates account for memoizing the Fibonacci numbers, the remaining updates result from lazy evaluation of the data structures used for memoization and only copy few bytes. Also, the garbage collection time for MemoCombinators is slightly increased.

Table 1. Comparison of the required runtime (left) and the copy effort (right) of computing the 150,000th Fibonacci number for the memoization libraries MemoTrie (MTrie) and MemoCombinators (MComb).

	Persistent Heap		**Volatile Heap**
	DRAM	PM	DRAM
MTrie	2,42 s	3,73 s	0,83 s
MComb	3,28 s	4,48 s	0,93 s

	thunk updates	**copy size**	**average bytes / thunk update**
MTrie	384 467	967 MB	2636,2 B
MComb	899 990	982 MB	1144,1 B

6 CONCLUSION AND FUTURE WORK

Persistent memory enables the direct modification of persistent data structures without serialization overhead. Providing transactions in persistent memory is challenging due to the small granularity of atomic updates (8 bytes). Although several transaction libraries are available, they require significant effort from programmers to use them. In this paper, we have demonstrated that a PSTM abstraction to interact with PM at the language level can be offered by extending Haskell's STM mechanism. Its simple API preserves the performance benefits of PM while hiding most of its details. PSTM is fully compatible with Haskell's lazy evaluation semantics, thus allowing easy adaptation of existing code. The evaluation of STM libraries transformed to support PSTM on real

hardware has shown that PSTM offers higher or the same performance compared to low-level library approaches. We further added persistence to two volatile memoization libraries. PSTM currently only supports single binary execution as raw function pointers are stored in PM. Future versions could be adapted to allow multiple binaries to interact with the same persistent memory root and support for multiple persistent roots could be introduced to further segment the persistence domain. Additionally, garbage collector pause times could be significantly reduced by allowing blocked threads to cooperate in nonmoving collection cycles, further increasing the throughput of the transaction mechanism.

ACKNOWLEDGMENTS

We would like to thank Simon Peyton-Jones and Jeremy Gibbons for the discussions that helped to develop the ideas on which our research is based. Thanks also to the community of the GHC IRC channel, which frequently provided very helpful support for compiler-related questions. Additionally, we are grateful for all the feedback and advice from the reviewers that helped to improve our work. Finally, we would like to thank the artifact reviewers for taking the time to reproduce our results and therefore to ensure the correctness and usability of our implementation.

REFERENCES

Martín Abadi, Luca Cardelli, Benjamin C. Pierce, and Gordon D. Plotkin. 1991. Dynamic Typing in a Statically Typed Language. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13, 2 (1991), 237–268. https://doi.org/10.1145/103135.103138

Malcolm P. Atkinson, Peter J. Bailey, Kenneth Chisholm, W. Paul Cockshott, and Ronald Morrison. 1983. An Approach to Persistent Programming. *Comput. J.* 26, 4 (1983), 360–365. https://doi.org/10.1093/comjnl/26.4.360

Malcolm P. Atkinson, Kenneth Chisholm, and W. Paul Cockshott. 1982. PS-algol: an Algol with a persistent heap. *ACM SIGPLAN Notices* 17, 7 (1982), 24–31. https://doi.org/10.1145/988376.988378

Malcolm P. Atkinson and Ronald Morrison. 1995. Orthogonally Persistent Object Systems. *VLDB J.* 4, 3 (1995), 319–401. http://www.vldb.org/journal/VLDBJ4/P319.pdf

Ömer S. Ağacan. 2020. *The problem with adding functions to compact regions.* Retrieved February 24, 2021 from https://www.well-typed.com/blog/2020/03/functions-in-compact-regions/

Jost Berthold. 2010. Orthogonal Serialisation for Haskell. In *Implementation and Application of Functional Languages - 22nd International Symposium, IFL 2010, Alphen aan den Rijn, The Netherlands, September 1-3, 2010, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 6647)*. Springer, 38–53. https://doi.org/10.1007/978-3-642-24276-2_3

Luc Bläser. 2007. Persistent Oberon: A Programming Language with Integrated Persistence. In *5th Asian Symposium on Programming Languages and Systems (APLAS), Singapore, November 29-December 1*, Vol. 4807. 71–85. https://doi.org/10.1007/978-3-540-76637-7_6

Peter T. Breuer. 2003. A Formal Model for the Block Device Subsystem of the Linux Kernel. In *Formal Methods and Software Engineering, Proceedings of the 5th International Conference on Formal Engineering Methods (ICFEM), Singapore, November 5-7*. 599–619. https://doi.org/10.1007/978-3-540-39893-6_34

Trevor Brown and Hillel Avni. 2016. PHyTM: Persistent Hybrid Transactional Memory. *Proceedings of the VLDB Endowment* 10, 4 (2016), 409–420. https://doi.org/10.14778/3025111.3025122

James Cheney and Ralf Hinze. 2002. A Lightweight Implementation of Generics and Dynamics. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*. Association for Computing Machinery, New York, NY, USA, 90–104. https://doi.org/10.1145/581690.581698

Paul Chiusano. 2020. *How Unison reduces ecosystem churn.* Retrieved February 24, 2021 from https://www.unisonweb.org/2020/04/10/reducing-churn/

Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. 2011. NV-Heaps: making persistent objects fast and safe with next-generation, non-volatile memories. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Newport Beach, CA, USA, March 5-11*. 105–118. https://doi.org/10.1145/1950365.1950380

Richard C. H. Connor. 1991. *Types and polymorphism in persistent programming systems.* Ph.D. Dissertation. University of St Andrews.

Alberto G. Corona. 2017. *TCache: A Transactional cache with user-defined persistence.* Retrieved February 24, 2021 from http://hackage.haskell.org/package/TCache

Andreia Correia, Pascal Felber, and Pedro Ramalhete. 2018. Romulus: Efficient Algorithms for Persistent Transactional Memory. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures (SPAA), Vienna, Austria, July 16-18, 2018*. ACM, 271–282. https://doi.org/10.1145/3210377.3210392

Rene De La Briandais. 1959. File Searching Using Variable Length Keys. In *Papers Presented at the the March 3-5, 1959, Western Joint Computer Conference (IRE-AIEE-ACM '59 (Western))*. ACM, San Francisco, California, 295–298. https://doi.org/10.1145/1457838.1457895

Subramanya Rao Dulloor, Sanjay Kumar, Anil S. Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. 2014. System software for persistent memory. In *9th Eurosys Conference, Amsterdam, The Netherlands, April 13-16*. 15:1–15:15. https://doi.org/10.1145/2592798.2592814

Conal Elliot. 2008. *Elegant memoization with functional memo tries*. Retrieved February 24, 2021 from http://conal.net/blog/posts/elegant-memoization-with-functional-memo-tries

Pascal Felber, Christof Fetzer, and Torvald Riegel. 2008. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), Salt Lake City, UT, USA, February 20-23*. 237–246. https://doi.org/10.1145/1345206.1345241

Richard F. Freitas and Winfried W. Wilcke. 2008. Storage-class memory: The next storage system technology. *IBM Journal of Research & Development* 52, 4-5 (2008), 439–448. https://doi.org/10.1147/rd.524.0439

Ben Gamari and Ömer Sinan Ağacan. 2019. *COMPACT_NFDATA support for the nonmoving collector*. Retrieved February 24, 2021 from https://gitlab.haskell.org/ghc/ghc/commit/77341c734f6c619d281c308b58a0f59b0b032aa2

Ben Gamari and Laura Dietz. 2020. Alligator collector: a latency-optimized garbage collector for functional programming languages. In *ACM SIGPLAN International Symposium on Memory Management (ISMM), June 16, 2020*. 87–99. https://doi.org/10.1145/3381898.3397214

GHC Team. 2020a. *The Block Allocator*. Retrieved February 24, 2021 from https://gitlab.haskell.org/ghc/ghc/-/wikis/commentary/rts/storage/block-alloc

GHC Team. 2020b. *GHC Commentary: The Layout of Heap Objects*. Retrieved February 24, 2021 from https://gitlab.haskell.org/ghc/ghc/-/wikis/commentary/rts/storage/heap-objects

Justin E. Gottschlich and Daniel A. Connors. 2007. DracoSTM: A Practical C++ Approach to Software Transactional Memory. In *Proceedings of the 2007 Symposium on Library-Centric Software Design, Montreal, Canada, October 21*. 52–66. https://doi.org/10.1145/1512762.1512768

Frank T. Hady, Annie P. Foong, Bryan Veal, and Dan Williams. 2017. Platform Storage Performance With 3D XPoint Technology. *Proc. IEEE* 105, 9 (2017), 1822–1833. https://doi.org/10.1109/JPROC.2017.2731776

Theo Haerder and Andreas Reuter. 1983. Principles of Transaction-Oriented Database Recovery. *ACM Comput. Surv.* 15, 4 (1983), 287–317. https://doi.org/10.1145/289.291

Cordelia V. Hall, Kevin Hammond, Will Partain, Simon L. Peyton Jones, and Philip Wadler. 1992. The Glasgow Haskell Compiler: A Retrospective. In *Proceedings of the Glasgow Workshop on Functional Programming, Ayr, Scotland, UK, 6-8 July*. 62–71. https://doi.org/10.1007/978-1-4471-3215-8_6

Robert Harper. 1985. Modules and Persistence in Standard ML. In *Data Types and Persistence. Edited Papers from the Proceedings of the First Workshop on Persistent Objects, Appin, Scotland, UK, August 1985 (Topics in Information Systems)*, Malcolm P. Atkinson, Peter Buneman, and Ronald Morrison (Eds.). Springer, 21–30.

Tim Harris, Simon Marlow, Simon L. Peyton Jones, and Maurice Herlihy. 2005. Composable memory transactions. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), June 15-17, 2005, Chicago, IL, USA*. 48–60. https://doi.org/10.1145/1065944.1065952

Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer III. 2003. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22nd ACM Symposium on Principles of Distributed Computing (PODC), Boston, Massachusetts, USA, July 13-16*. 92–101. https://doi.org/10.1145/872035.872048

Maurice Herlihy and J. Eliot B. Moss. 1993. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA), San Diego, CA, USA, May 1993*. 289–300. https://doi.org/10.1145/165123.165164

Antony L. Hosking and Jiawan Chen. 1999. PM3: An Orthogonal Persistent Systems Programming Language - Design, Implementation, Performance. In *Proceedings of 25th International Conference on Very Large Data Bases (VLDB), September 7-10, Edinburgh, Scotland, UK*. 587–598.

Paul Hudak, Simon L. Peyton Jones, Philip Wadler, Brian Boutel, Jon Fairbairn, Joseph H. Fasel, María M. Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, Richard B. Kieburtz, Rishiyur S. Nikhil, Will Partain, and John Peterson. 1992. Report on the Programming Language Haskell, A Non-strict, Purely Functional Language. *ACM SIGPLAN Notices* 27, 5 (1992). https://doi.org/10.1145/130697.130699

Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. 2019. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. *CoRR* abs/1903.05714 (2019). http://arxiv.org/abs/1903.05714

Alfons Kemper and Donald Kossmann. 1993. Adaptable Pointer Swizzling Strategies in Object Bases. In *Proceedings of the Ninth International Conference on Data Engineering, April 19-23, 1993, Vienna, Austria*. IEEE Computer Society, 155–162. https://doi.org/10.1109/ICDE.1993.344067

Emre Kultursay, Mahmut T. Kandemir, Anand Sivasubramaniam, and Onur Mutlu. 2013. Evaluating STT-RAM as an energy-efficient main memory alternative. In *2012 IEEE International Symposium on Performance Analysis of Systems & Software, Austin, TX, USA, 21-23 April, 2013*. IEEE Computer Society, 256–267. https://doi.org/10.1109/ISPASS.2013.6557176

Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. 2017. DudeTM: Building Durable Transactions with Decoupling for Persistent Memory. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Xi'an, China, April 8-12*. 329–343. https://doi.org/10.1145/3037697.3037714

Simon Marlow, Tim Harris, Roshan P. James, and Simon L. Peyton Jones. 2008. Parallel generational-copying garbage collection with a block-structured heap. In *Proceedings of the 7th International Symposium on Memory Management (ISMM), Tucson, AZ, USA, June 7-8*. 11–20. https://doi.org/10.1145/1375634.1375637

Alonso Marquez, Stephen Blackburn, Gavin Mercer, and John N. Zigman. 2000. Implementing Orthogonally Persistent Java. In *Persistent Object Systems, 9th International Workshop, POS-9, Lillehammer, Norway, September 6-8, 2000, Revised Papers (Lecture Notes in Computer Science, Vol. 2135)*, Graham N. C. Kirby, Alan Dearle, and Dag I. K. Sjøberg (Eds.). Springer, 247–261. https://doi.org/10.1007/3-540-45498-5_22

Markus Mäsker, Tim Süß, Lars Nagel, Lingfang Zeng, and André Brinkmann. 2019. Hyperion: Building the Largest In-memory Search Tree. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD), Amsterdam, The Netherlands, June 30 - July 5*. 1207–1222. https://doi.org/10.1145/3299869.3319870

David J. McNally and Antony J. T. Davie. 1991. Two Models For Integrating Persistence and Lazy Functional Languages. *ACM SIGPLAN Notices* 26, 5 (1991), 43–52. https://doi.org/10.1145/122501.122504

Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. 2008. STAMP: Stanford Transactional Applications for Multi-Processing. In *4th International Symposium on Workload Characterization (IISWC), Seattle, Washington, USA, September 14-16, 2008*. IEEE Computer Society, 35–46. https://doi.org/10.1109/IISWC.2008.4636089

Ron Morrison, Richard Connor, Graham Kirby, David Munro, Malcolm Atkinson, Quintin Cutts, Alfred Brown, and Alan Dearie. 2000. *The Napier88 Persistent Programming Language and Environment*. 98–154. https://doi.org/10.1007/978-3-642-59623-0_6

Nafiseh Moti, Frederic Schimmelpfennig, Reza Salkhordeh, David Klopp, Toni Cortes, Ulrich Rückert, and André Brinkmann. 2021. Simurgh: A Fully Decentralized and Secure NVMM User Space File System. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC) (accepted for publication), St. Louis, MO, USA, November 14-19, 2021*.

Sanketh Nalli. 2018. *GCC port of TM system Mnemosyne*. Retrieved February 24, 2021 from https://github.com/snalli/mnemosyne-gcc

Luke Palmer. 2013. *Data.MemoCombinators: Combinators for building memo tables*. Retrieved February 24, 2021 from https://hackage.haskell.org/package/data-memocombinators

Stuart S. P. Parkin, Masamitsu Hayashi, and Luc Thomas. 2008. Magnetic Domain-Wall Racetrack Memory. *Science* 320, 5873 (2008), 190–194.

Simon L. Peyton Jones. 1992. Implementing Lazy Functional Languages on Stock Hardware: The Spineless Tagless G-Machine. *Journal of Functional Programming* 2, 2 (1992), 127–202. https://doi.org/10.1017/S0956796800000319

Simon L. Peyton Jones, Andrew D. Gordon, and Sigbjørn Finne. 1996. Concurrent Haskell. In *23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), St. Petersburg Beach, Florida, USA, January 21-24*. 295–308. https://doi.org/10.1145/237721.237794

Juan J. Quintela and Juan J. Sánchez. 2001. Persistent Haskell. In *Computer Aided Systems Theory - EUROCAST 2001, Las Palmas de Gran Canaria, Spain, February 19-23, 2001, Revised Papers (Lecture Notes in Computer Science, Vol. 2178)*, Roberto Moreno-Díaz, Bruno Buchberger, and José Luis Freire (Eds.). Springer, 657–667. https://doi.org/10.1007/3-540-45654-6_50

Pedro Ramalhete, Andreia Correia, Pascal Felber, and Nachshon Cohen. 2019. OneFile: A Wait-Free Persistent Transactional Memory. In *49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Portland, OR, USA, June 24-27, 2019*. IEEE, 151–163. https://doi.org/10.1109/DSN.2019.00028

Dulloor Subramanya Rao, Sanjay Kumar, Anil S. Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. 2014. System software for persistent memory. In *Ninth Eurosys Conference 2014, EuroSys 2014, Amsterdam, The Netherlands, April 13-16, 2014*, Dick C. A. Bulterman, Herbert Bos, Antony I. T. Rowstron, and Peter Druschel (Eds.). ACM, 15:1–15:15. https://doi.org/10.1145/2592798.2592814

Simone Raoux, Geoffrey W. Burr, Matthew J. Breitwisch, Charles T. Rettner, Yi-Chou Chen, Robert M. Shelby, Martin Salinga, Daniel Krebs, Shih-Hung Chen, Hsiang-Lan Lung, and Chung Hon Lam. 2008. Phase-change random access memory: A scalable technology. *IBM Journal of Research and Development* 52, 4-5 (2008), 465–480.

Peter Robinson. 2019. *concurrent-hashtable: Thread-safe hash tables for multi-cores!* Retrieved February 24, 2021 from http://hackage.haskell.org/package/concurrent-hashtable

Andy Rudoff. 2017. Persistent Memory Programming. *login Usenix Mag.* 42, 2 (2017). https://www.usenix.org/publications/login/summer2017/rudoff

Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Ben Hertzberg. 2006. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), New York, New York, USA, March 29-31*. 187–197. https://doi.org/10.1145/1122971.1123001

Thomas Shull, Jian Huang, and Josep Torrellas. 2019. AutoPersist: an easy-to-use Java NVM framework based on reachability. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Phoenix, AZ, USA, June 22-26*. 316–332. https://doi.org/10.1145/3314221.3314608

Christopher Strachey. 2000. Fundamental Concepts in Programming Languages. *High. Order Symb. Comput.* 13, 1/2 (2000), 11–49. https://doi.org/10.1023/A:1010000313106

Robert D. Tennent. 1977. Language Design Methods Based on Semantic Principles. *Acta Informatica* 8 (1977), 97–112. https://doi.org/10.1007/BF00289243

Katsuhiro Ueno and Atsushi Ohori. 2016. A fully concurrent garbage collector for functional programs on multicore processors. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP), Nara, Japan, September 18-22*. 421–433. https://doi.org/10.1145/2951913.2951944

Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: lightweight persistent memory. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Newport Beach, CA, USA, March 5-11*. 91–104. https://doi.org/10.1145/1950365.1950379

Michèle Weiland, Holger Brunst, Tiago Quintino, Nick Johnson, Olivier Iffrig, Simon D. Smart, Christian Herold, Antonino Bonanni, Adrian Jackson, and Mark Parsons. 2019. An early evaluation of Intel's optane DC persistent memory module and its impact on high-performance scientific applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC), Denver, Colorado, USA, November 17-19*. 76:1–76:19. https://doi.org/10.1145/3295500.3356159

Mingyu Wu, Haibo Chen, Hao Zhu, Binyu Zang, and Haibing Guan. 2020. GCPersist: an efficient GC-assisted lazy persistency framework for resilient Java applications on NVM. In *VEE '20: 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, virtual event, Lausanne, Switzerland, March 17*. 1–14. https://doi.org/10.1145/3381052.3381318

Mingyu Wu, Ziming Zhao, Haoyu Li, Heting Li, Haibo Chen, Binyu Zang, and Haibing Guan. 2018. Espresso: Brewing Java For More Non-Volatility with Non-volatile Memory. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Williamsburg, VA, USA, March 24-28*. 70–83. https://doi.org/10.1145/3173162.3173201

Jian Xu and Steven Swanson. 2016. NOVA: A Log-Structured File System for Hybrid Volatile/Non-Volatile Main Memories. *login Usenix Mag.* 41, 3 (2016). https://www.usenix.org/publications/login/fall2016/xu

Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy Rudoff. 2017. NOVA-Fortis: A Fault-Tolerant Non-Volatile Main Memory File System. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP), Shanghai, China, October 28-31*. 478–496. https://doi.org/10.1145/3132747.3132761

Edward Z. Yang, Giovanni Campagna, Ömer S. Agacan, Ahmed El-Hassany, Abhishek Kulkarni, and Ryan R. Newton. 2015. Efficient communication and collection with compact normal forms. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*. ACM, 362–374. https://doi.org/10.1145/2784731.2784735

Ryan Yates and Michael L. Scott. 2017. Improving STM performance with transactional structs. In *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell, Oxford, United Kingdom, September 7-8, 2017*. ACM, 186–196. https://doi.org/10.1145/3122955.3122972

Ryan Yates and Michael L. Scott. 2019. Leveraging hardware TM in Haskell. In *Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), Washington, DC, USA, February 16-20, 2019*. ACM, 94–106. https://doi.org/10.1145/3293883.3295711