# Recoverable Software Combining*

Panagiota Fatourou†
Université de Paris, LIPADE, F-75006
Paris, France
FORTH and University of Crete
Greece
faturu@csd.uoc.gr

Nikolaos D. Kallimanis
Institute of Computer Science,
Foundation for Research &
Technology - Hellas (FORTH)
Greece
nkallima@ics.forth.gr

Eleftherios Kosmas‡
Department of Computer Science,
University of Crete
Greece
ekosmas@csd.uoc.gr

## 1 INTRODUCTION

Byte-addressable Non-Volatile Main Memory (NVMM) is a reality, with Intel Optane DC Persistent Memory being already in the market. The availability of NVMM enables the design of concurrent algorithms, whose execution will be recoverable at low cost. An algorithm is *recoverable* if its state can be restored after recovery from a system-crash failure. Despite many efforts for designing efficient recoverable synchronization protocols and data structures, persistence comes at a significant cost even for fundamental data structures, such as stacks and queues. The main reason for this is that data stored into registers and caches are volatile, and need to be flushed to persistent memory for ensuring that they will not be lost at a system crash.

*Software combining* is a state-of-the-art synchronization technique [5, 6] which works well when the number of synchronization points in the underlying algorithm is small. In this paper, we reveal the power of software combining in achieving recoverable synchronization and designing recoverable data structures. In software combining, each thread first announces its request, and then tries to become the combiner by acquiring a lock. The combiner applies several active requests, in addition to its own, before it releases the lock. As long as the combiner serves active requests, other threads perform local spinning, waiting for the combiner to release the lock. As soon as the lock is released, waiting threads whose requests have been served by the combiner, return the calculated responses, whereas the rest compete again for the lock.

In this paper, we present two *recoverable* software combining protocols, which have been designed carefully to respect a number of principles we identify to be crucial for performance and they can be summarized as follows: 1) Store in NVMM only those variables that are absolutely necessary for recoverability to minimize the number of expensive persistence instructions, such as flushes (pwbs) and fences (psyncs) that are executed; 2) As not all all persistence instructions have the same cost [1], design the protocol so that the persistence instructions it executes are of low cost. One way to achieve this is to avoid the execution of such instructions on highly-contended shared variables; and 3) place the data to be persisted in consecutive memory addresses and persist them all together. Our experiments show that the resulting protocols are many times faster than a large collection of existing recoverable techniques for achieving scalable synchronization.

State-of-the-art combining protocols [6] do not necessarily support persistence in an efficient way, as they often store the active requests in scattered memory locations (and in particular, in a dynamic singly-linked list). The combiner traverses these locations to discover the currently active requests and applies them on the shared state of the object, recording their responses in the nodes of the list. As in such a protocol, the data are scattered in memory, some of the persistence principles mentioned above are violated, leading to high persistence overhead. Our synchronization protocols have been designed to respect all persistence principles. We build recoverable queues and stacks using our protocols. Experiments show that the proposed implementations outperform by far, many existing synchronization techniques [3, 4, 10], as well as recoverable data structures based on such techniques and specialized recoverable implementations of them [8, 11, 12]. Concurrent queues and stacks play a significant role in runtime systems, high performance computing, kernel schedulers, network interfaces, etc. The proliferation of NVMM and the availability of highly-efficient recoverable stacks and queues could enable persistence in such settings. Our implementations satisfy *detectable recoverability* [8], whereas most competitors guarantee only weaker consistency properties [9].

Our contributions can be summarized as follows: 1) We present two highly-efficient recoverable combining protocols (called PBcomb and PWFcomb), which ensure low persistence overhead than previous algorithms; 2) The protocols perform much better than state-of-the-art existing synchronization techniques; 3) We provide a list of persistence principles that are crucial for performance and provide experiments to illustrate the performance power of respecting these principles when designing synchronization protocols; 4) We provide recoverable queues (PBqueue and PWFqueue) and stacks (PBstack and PWFstack), based on the new protocols, which are much more efficient than previous recoverable implementations of such data structures.

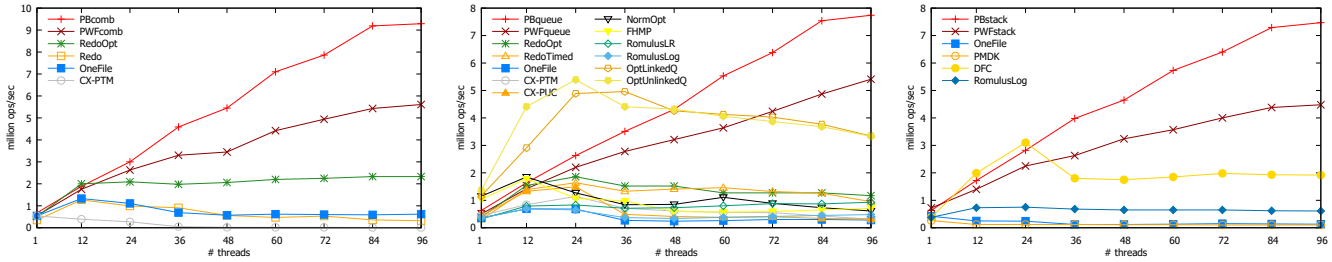
## 2 OUR PROTOCOLS

PBcomb implements the lock in volatile memory using an implementation which has low synchronization cost. Additionally, a thread may leave the entry-section without ever acquiring the lock, if it finds out that its request has been served by a combiner. Also, PBcomb utilizes an array, *Request*, to ensure that active requests are stored in consecutive memory addresses. This array is stored

*These results have been accepted in ACM PPoPP 2022 [7].

†Supported by the EU Horizon 2020, Marie Sklodowska-Curie project with GA No 101031688

‡This research is co-financed by Greece and the European Union (European Social Fund- ESF) through the Operational Programme «Human Resources Development, Education and Lifelong Learning» in the context of the project "Reinforcement of Postdoctoral Researchers - 2nd Cycle" (MIS-5033021), implemented by the State Scholarships Foundation (IKY)




(a) (b) (c)

**Figure 1: Throughput of recoverable: (a) AtomicFloat objects, (b) queue implementations, and (c) stack implementations.**

in volatile memory, resulting in lower persistence cost. Each combiner creates a copy of the state of the implemented object and applies the active requests on this private copy. This is a crucial design decision of PBcomb in terms of performance. The combiner switches a shared variable to index its copy, indicating that it stores the current valid state of the implemented object. The combiner should persist its private copy before trying to switch the pointer.

When a combiner performs updates directly on the shared state of the implemented object the updates are performed on data that are usually scattered in memory, thus resulting in high persistence cost when persisting the updated values. This problem is avoided by the technique of creating a copy of the state to apply the updates on, followed by PBcomb. This technique allows to persist data stored in the copy in consecutive memory addresses but it works well mainly for objects of small or medium size (or when the number of synchronization points is small), as otherwise the cost of copying and persisting the state may dominate the cost of persisting a smaller amount of scattered data (part of the state).

To be compatible with the persistence principles listed above, PBcomb stores the response values in an array, which is maintained together with the state of the object (in consecutive memory addresses). The combiner persists the entire array of return values together with the object's state. PBcomb uses two bits (*activate* and *deactivate*) for each thread $p$, to identify whether the last request, initiated by $p$, has been served. If the two bits are not equal, $p$ has a request which has not yet been served i.e., it is *active*. PBcomb persists just the deactivate bit of $p$. Following persistence principle 3, PBcomb stores the deactivate bits together with the object's state, so all data to be persisted are in consecutive memory locations.

Details for the other implementations are provided in [7].

## 3 EVALUATION

We evaluate our algorithms on a 48-core machine (96 logical cores) consisting of 2 Intel Xeon Platinum 8260M processors with 24 cores each, equipped with a 1TB Intel Optane DC persistent memory (DCPMM). Each run simulates $10^7$ atomic operations in total, with each of the $n$ threads simulating $10^7/n$ operations.

We first consider a synthetic benchmark (`AtomicFloat`) in which every thread, repeatedly, executes `AtomicFloat`($O$, $k$) that reads the value $v$ of $O$ and updates it to $v * k$. In Figure 1a, we compare the performance of `AtomicFloat` implementations based on PBcomb and PWFcomb against state-of-the-art wait-free recoverable synchronization techniques: OneFile [10], CX-PTM [4], and RedoOpt [4]. Figure 1a shows that our first protocol, PBcomb, is more than 4x faster than RedoOpt, which is the fastest among the competitors. Also, our second protocol, PWFcomb is more than 2.8x faster than RedoOpt. Figure 1b compares the performance of PBqueue and PWFqueue with recoverable queue implementations based on the persistence techniques studied in Figure 1a and CX-PUC [4], the specialized recoverable queue implementations in [8] (FHMP) and in [12] (OptLinkedQ and OptUnlinkedQ), an implementation based on Capsules-Normal [2] (NormOpt), and the recoverable queue implementations based on Romulus [3] (i.e., RomulusLR and RomulusLog). Figure 1b shows that PBqueue achieves superior performance by being 2x faster than the OptUnlinkedQ, which is the best competitor. Finally, Figure 1c illustrates that the performance of PBstack and PWFstack is much better than the following algorithms: the recoverable stack implementations based on OneFile [10] and Romulus [3], and a recoverable stack based on flat-combining (DFC) [11], which is the best competitor.

## REFERENCES


[1] H. Attiya, O. Ben-Baruch, P. Fatourou, D. Hendler, and E. Kosmas. Detectable recovery of lock-free data structures. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '22, to appear.

[2] N. Ben-David, G. E. Blelloch, M. Friedman, and Y. Wei. Delay-free concurrency on faulty persistent memory. In *The 31st ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '19, pages 253–264.

[3] A. Correia, P. Felber, and P. Ramalhete. Romulus: Efficient algorithms for persistent transactional memory. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, SPAA '18, pages 271–282.

[4] A. Correia, P. Felber, and P. Ramalhete. Persistent memory and the rise of universal constructions. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020.

[5] P. Fatourou and N. D. Kallimanis. A highly-efficient wait-free universal construction. In *Proceedings of the Twenty-Third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, pages 325–334.

[6] P. Fatourou and N. D. Kallimanis. Revisiting the combining synchronization technique. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 257–626.

[7] P. Fatourou, N. D. Kallimanis, and E. Kosmas. The performance power of software combining in persistence. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '22, to appear.

[8] M. Friedman, M. Herlihy, V. Marathe, and E. Petrank. A persistent lock-free queue for non-volatile memory. *ACM SIGPLAN Notices*, 53(1):28–40, 2018.

[9] J. Izraelevitz, H. Mendes, and M. L. Scott. Linearizability of persistent memory objects under a full-system-crash failure model. In *Proceedings of the 30th International Symposium of Distributed Computing*, volume LNCS 9888 of *DISC '16*, pages 313–327.

[10] P. Ramalhete, A. Correia, P. Felber, and N. Cohen. Onefile: A wait-free persistent transactional memory. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 151–163. IEEE, 2019.

[11] M. Rusanovsky, O. Ben-Baruch, D. Hendler, and P. Ramalhete. A flat-combining-based persistent stack for non-volatile memory. *CoRR*, abs/2012.12868, 2020 (version submited at 23 December, 2020).

[12] G. Sela and E. Petrank. Durable queues: The second amendment. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures*, pages 385–397.