Brief Announcement: Persistent Software Combining

Panagiota Fatourou
Foundation for Research and Technology – Hellas, Institute of Computer Science, Heraklion, Greece
University of Crete, Department of Computer Science, Heraklion, Greece

Nikolaos D. Kallimanis
Foundation for Research and Technology – Hellas, Institute of Computer Science, Heraklion, Greece

Eleftherios Kosmas
University of Crete, Department of Computer Science, Heraklion, Greece

Abstract

We study the performance power of software combining in designing recoverable algorithms and data structures. We present two recoverable synchronization protocols, one blocking and another wait-free, which illustrate how to use software combining to achieve both low persistence and synchronization cost. Our experiments show that these protocols outperform by far state-of-the-art recoverable universal constructions and transactional memory systems. We built recoverable queues and stacks, based on these protocols, that exhibit much better performance than previous such implementations.

2012 ACM Subject Classification Theory of computation → Concurrent algorithms; Theory of computation → Data structures design and analysis

Keywords and phrases Persistent objects, recoverable algorithms, durability, synchronization protocols, software combining, universal constructions, wait-freedom, stacks, queues

Digital Object Identifier 10.4230/LIPIcs.DISC.2021.56

Related Version *Full Version*: `https://arxiv.org/abs/2107.03492`

Funding *Panagiota Fatourou*: Supported by the EU Horizon 2020, Marie Sklodowska-Curie project with GA No 101031688. *Eleftherios Kosmas:* 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).

1 Motivation and Contribution

The availability of byte-addressable non-volatile main memory (NVMM) enables the design of recoverable concurrent algorithms. An algorithm is recoverable (also known as *persistent*) if its state can be restored after recovery from a *system-crash* failure. Another important property, known as *detectability* [15], is to be able to determine, upon recovery, if a request has been completed, and if yes, to find its response.

When designing recoverable algorithms, the main challenge stems from the fact that all data stored into registers and caches are volatile. Thus, unless they have been flushed to persistent memory, such data will be lost at a system crash. Flushing to persistent memory occurs by including specific *persistence instructions*, such as `pwb`, `pfence` and `psync` in the code, which are however expensive in terms of performance. Despite many efforts for designing efficient recoverable synchronization protocols and data structures, persistence comes at an important cost even for fundamental data structures, such as queues and stacks.

We provide recoverable implementations of software combining protocols that exhibit much better performance and lower persistence cost, in comparison to a large collection of existing persistent techniques [3, 4, 22, 23] for achieving scalable synchronization. The implementation

of many synchronization protocols, including queue-lock implementations [5, 19, 20] and software combining protocols [8, 9, 10, 11, 16, 21], require careful design to maintain them simple and low-cost; even minor changes in their code could negatively impact performance, rendering them uninteresting. This was also the case with our protocols. Coming up with them was quite intricate and they required careful design to ensure good performance.

As a recoverable implementation often inherents the synchronization overheads of the concurrent implementation it is inspired from, persistence should not be added on top of heavy universal constructions or transactional memory (TM) systems with high synchronization cost. To achieve this goal, we first design a new combining protocol, called BComb, that outperforms, by far, not only previous state-of-the art conventional combining protocols [8, 9, 10, 16], but also NUMA-aware synchronization techniques [6, 9]. Maintaining the number of persistence instructions low is also very important in terms of performance. We present PBComb, a recoverable blocking combining protocol built upon BComb, that reveals the power of appropriately utilizing software combining in achieving very low peristence cost.

We also present a wait-free recoverable universal construction, called PWFComb. PWFComb combines and extends ideas from PBComb and PSim [8, 10]. PSim [8, 10] is a state-of-the-art wait-free universal construction [7, 12, 13], which has been designed as the practical version of Herlihy's universal construction [17]. Our experimental analysis shows that both PBComb and PWFComb outperform by far their competitors [3, 4, 22, 23].

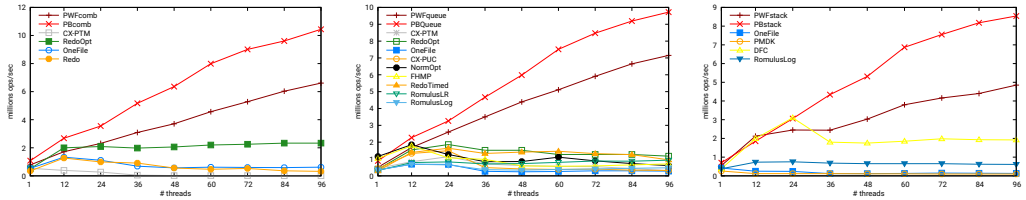
2 PBCOMB and PWFCOMB

Brief Description. PBComb follows the general idea of blocking software combining [9, 16, 21]. Each thread first announces its request and then tries to become the combiner. PBComb uses an array for storing the announced requests and a lock to identify which thread will become the combiner. After acquiring the lock, the combiner creates a copy of the state of the simulated object and applies the active requests on this copy. Then, the combiner switches a shared variable to index this copy, indicating that it stores the current valid state of the simulated object, and releases the lock. As long as the combiner serves active requests, other active threads perform local spinning, waiting for the combiner to release the lock.

PBComb owes its low persistence cost to the following two achievements: i) threads other than the combiner do not have to execute any persistence instruction, and ii) the combiner does not persist each of the requests it applies separately; it persists all the variables it access together, after it is done with the simulation of all requests it serves. Let the *combining degree* d be the average number of requests that a combiner serves. PBComb executes a small number of `pwb` instructions for every d requests. Achieving this, on top of a light-weight synchronization protocol, are the main reasons that PBComb has very good performance.

In PWFComb, many threads may simultaneously attempt to be the combiner to achieve wait-freedom: they copy the state of the object locally and use this local copy to apply all active requests they see announced. Then, each of them attempts to change a pointer S to point to its own local copy using SC. Only a single thread among them (the *combiner*) manages to do so. Achieving this form of wait-freedom using naive persistence approaches, would result in significant persistence overhead, as several threads attempt to become the combiner, and thus they all have to persist certain variables to ensure durability without blocking each other. Thus, we came up with more intricate persistence schemes to reduce this cost. PWFComb outperforms by far all competitors. However, it has worse performance than PBComb which pays less persistence overhead and is more light-weight.

Experimental Analysis. We performed our experimental analysis in a 48-core machine (96 logical cores) consisting of 2 Intel Xeon Platinum 8260M processors, each of which consists of 24 cores. The machine is equipped with a 1TB Intel Optane DC persistent memory



(a) (b) (c)

Figure 1 a) Average throughput of implementations while simulating a recoverable `AtomicFloat` object. b)-c) Average throughput of recoverable queue and stack implementations, respectively.

(DCPMM) (configured in AppDirect mode). Figure 1a shows that both, PBcomb and PWFcomb, outperform by far, many previous recoverable TM systems. Specifically, we compare the performance of PBcomb and PWFcomb against the following algorithms: OneFile [23], CX-PUC [4], CX-PTM [4], and RedoOpt [4]. For our experiments, we used the latest version of code for these algorithms that is provided in [2]. The diagrams show that PBcomb is 4x faster and PWFcomb is 2.8x faster than the competitors. Our protocols satisfy *detectability* [15] and are wait-free, whereas most competitors guarantee only weaker consistency (such as *durable linearizability* [18]), and/or progress.

Figures 1b and 1c illustrate that the recoverable queues (PBqueue and PWFqueue) and stacks (PBstack and PWFstack) that are built on top of PBcomb and PWFcomb, have much better performance than state-of-the-art recoverable implementations of such data structures, including the specialized recoverable queue implementations in [15]. Specifically, Figure 1b compares PBqueue and PWFqueue with the specialized recoverable queue implementation (FHMP) by Friedman *et al.* [15], the recoverable queue implementation (NormOpt) based on Capsules-Normal [1], and recoverable queue implementations based on Romulus [3] (RomulusLR and RomulusLog). Figure 1b shows that PWFqueue and PBqueue achieve superior performance, with PBqueue being more than $5x$ faster than the queue based on RedoOpt, which is the best competitor. Finally, Figure 1c compares the performance of PBstack and PWFstack against recoverable stack implementations based on OneFile [23] and Romulus [3], and against the archived recoverable stack based on flat-combining (DFC) [24]. PWFstack and PBstack exhibit much better performance than all the competitors. Specifically, PBstack is $4.4x$ faster than the DFC recoverable stack, which is the best competitor. See [14] for a full version of our paper which provides more diagrams and justification for the good performance of our algorithms.

3 Conclusion

The contributions of this paper can be summarized as follows.

- We present highly efficient recoverable software combining protocols that outperform *by far* many state-of-the-art recoverable universal constructions and TM systems.
- We built recoverable queues and stacks, based on our software combining protocols, which significantly outperform previous recoverable implementations of stacks and queues.
- Our results reveal the power of software combining in designing low cost persistent synchronization protocols and concurrent data structures.

References

1 N. Ben-David, G. E. Blelloch, M. Friedman, and Y. Wei. Delay-free concurrency on faulty persistent memory. In *ACM Symp. on Parallelism in Algorithms and Architectures (SPAA)*, pages 253–264, 2019.

2 A. Correia, P. Felber, and P. Ramalhete. The Code for RedoDB. URL: `https://github.com/pramalhe/RedoDB`.

3 A. Correia, P. Felber, and P. Ramalhete. Romulus: Efficient algorithms for persistent transactional memory. In *Symp. on Parallelism in Algorithms and Architectures (SPAA)*, pages 271–282, 2018.

4 A. Correia, P. Felber, and P. Ramalhete. Persistent memory and the rise of universal constructions. In *European Conference on Computer Systems (EuroSys)*, 2020.

5 T. S. Craig. Building FIFO and priority-queueing spin locks from atomic swap. Technical Report TR 93-02-02, Dept. of Computer Science, University of Washington, 1993.

6 D. Dice, V. J. Marathe, and N. Shavit. Lock cohorting: a general technique for designing numa locks. *ACM SIGPLAN Notices*, 47(8):247–256, 2012.

7 P. Fatourou and N. D. Kallimanis. The RedBlue Adaptive Universal Constructions. In *International Symp. on Distributed Computing (DISC)*, pages 127–141, 2009.

8 P. Fatourou and N. D. Kallimanis. A highly-efficient wait-free universal construction. In *ACM Symp. on Parallelism in Algorithms and Architectures (SPAA)*, pages 325–334, 2011.

9 P. Fatourou and N. D. Kallimanis. Revisiting the combining synchronization technique. In *ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)*, pages 257–626, 2012.

10 P. Fatourou and N. D. Kallimanis. Highly-efficient wait-free synchronization. *Theory of Computing Systems*, 55(3):475–520, 2014.

11 P. Fatourou and N. D. Kallimanis. Lock oscillation: Boosting the performance of concurrent data structures. In *Conf. on Principles of Distributed Systems (OPODIS)*, 2018.

12 P. Fatourou and N. D. Kallimanis. The RedBlue family of universal constructions. *Distributed Computing*, 33:485–513, 2020.

13 P. Fatourou, N. D. Kallimanis, and E. Kanellou. An efficient universal construction for large objects. In *Conf. on Principles of Distributed Systems (OPODIS)*, pages 18:1–18:15, 2018.

14 P. Fatourou, N. D. Kallimanis, and E. Kosmas. Persistent software combining. *CoRR*, abs/2107.03492, 2021. `arXiv:2107.03492`.

15 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.

16 D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *ACM Symp. on Parallelism in algorithms and architectures (SPAA)*, pages 355–364, 2010.

17 M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, 1993.

18 J. Izraelevitz, H. Mendes, and M. L. Scott. Linearizability of persistent memory objects under a full-system-crash failure model. In *International Symp. on Distributed Computing (DISC)*, pages 313–327, 2016.

19 P. S. Magnusson, A. Landin, and E. Hagersten. Queue Locks on Cache Coherent Multiprocessors. In *International Parallel Processing Symposium*, pages 165–171, 1994.

20 J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. on Computer Systems (TOCS)*, 9(1):21–65, 1991.

21 Y. Oyama, K. Taura, and A. Yonezawa. Executing parallel programs with synchronization bottlenecks efficiently. In *Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications (PDSIA)*, pages 182–204, 1999.

22 PMDK. The persistent memory development kit. URL: `https://github.com/pramalhe/RedoDB`.

23 P. Ramalhete, A. Correia, P. Felber, and N. Cohen. Onefile: A wait-free persistent transactional memory. In *IEEE Conf. on Dependable Systems and Networks (DSN)*, pages 151–163, 2019.

24 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). `arXiv:2012.12868`.