Tracking in Order to Recover — Detectable Recovery of Lock-Free Data Structures*

Hagit Attiya†
Technion
hagit@cs.technion.ac.il

Ohad Ben-Baruch†
Ben-Gurion University
ohadben@post.bgu.ac.il

Panagiota Fatourou
FORTH ICS &
University of Crete, CSD
faturu@csd.uoc.gr

Danny Hendler†
Ben-Gurion University
hendlerd@cs.bgu.ac.il

Eleftherios Kosmas‡
University of Crete, CSD
ekosmas@csd.uoc.gr

1 INTRODUCTION

Byte-addressable *non-volatile main memory* (*NVRAM*) combines the performance benefits of conventional main memory with the durability of secondary storage. Systems with NVRAM will be more prevalent in the near future. The availability of durable main memory has increased the interest in the *crash-recovery* model, in which failed processes may be resurrected after the system crashes. Of particular interest is the design of *recoverable concurrent data structures* (also called *persistent* [3] or *durable* [7]), whose operations can recover from crash-failures. Such data structures are important as they are building blocks for constructing simple, well-structured, sound and error-resistant multiprocessor systems. For example, in many big-data applications, shared in-memory tree-based data indices are created for fast data retrieval and useful data analytics.

When designing recoverable data structures, it is important to be able to tell after recovery whether an operation was executed to completion and if so, what its response was, a property called *detectable recovery* [1, 4]. In many computer systems (e.g., databases), detectable recovery is supported by precisely *logging* the progress of computations to non-volatile storage, and replaying the log during recovery. Logging imposes significant overheads in time and space. This cost is even more pronounced for concurrent data structures, where there is an extra cost of synchronizing log accesses.

We present the *Info Structure Based (ISB) tracking approach* for deriving *detectable* implementations of many widely-used concurrent data structures for systems with *non-volatile main memory* (*NVRAM*). ISB tracking avoids full-fledged logging, and tracks the progress of each operation individually, in a way supporting detectable recovery. Specifically, it explicitly maintains an *Info structure*, stored in non-volatile memory, to track an operation's progress as it executes. The Info structure allows a process to decide, upon recovery, whether the operation's effect has already become visible to other processes, in which case, the mechanism allows to determine the response of the operation. ISB-tracking is widely applicable—it can be used to derive recoverable versions of a large collection of concurrent data structures, including concurrent tree-like structures that could be used as indices.

In many cases, ISB-tracking requires small changes to the original code. It significantly saves on the cost (in both time and memory) incurred by tracking operations' progress, by not having to track which instructions have been performed exactly, but rather, specific stages of the operation. Furthermore, even this can often be piggybacked on information already tracked by the mechanisms that ensure progress in many existing concurrent data structures. These properties are what make our approach attractive.

We emphasize that detectability is a challenge even if caches are non-volatile, i.e., writes are immediately persisted, in program order. However, ISB-tracking informs how *persistency instructions* (flushes and fences) should be inserted for ensuring an implementation's correctness in an efficient manner, even when cache memories are volatile and their content is lost upon a system-wide failure [6].

Summarizing, the main contributions of this paper are: i) we propose ISB-tracking, a new mechanical transformation for deriving detectably recoverable implementations of concurrent data structures, ii) in a system with volatile caches, we present how persistency instructions can be added in ISB-tracking in a manner that enhances efficiency and scalability, iii) we apply ISB-tracking to get new detectably recoverable implementations of a wide collection of data structures, and iv) we provide an experimental analysis to compare ISB tracking with *all* existing relevant transformations and detectably recoverable concurrent data structures we are aware of. They show the feasibility of ISB-tracking and the good scalability it exhibits in many cases.

2 INFO-STRUCTURE BASED TRACKING

Many concurrent data structures employ a *helping mechanism* to ensure global progress (and specifically, a property known as *lock-freedom*), even if processes crash. Such implementations associate an information (*Info*) structure with each update, tracking the progress of the update by storing sufficient information to allow its completion by concurrent operations. In brief, this *Info-Structure-Based (ISB)* helping works as follows: An operation *Op* by process *q* initializes an Info structure and then attempts to *install* it in every node that *Op* is trying to change (as well as to those nodes that are needed to determine its response); this is done by executing a *CAS* to set a pointer in each of these nodes to point to the Info structure

*Full version available at: http://arxiv.org/abs/1905.13600.

†Supported in part by ISF grant 380/18.

‡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).

initialized by *Op*. Once the Info structure is successfully installed, q continues the execution of *Op* using the information stored in the Info structure. Once the update completes, *Op invalidates* the Info structure in all nodes pointed to it. If it fails to install the Info structure on some node, q finds the Info structure of another operation installed at the node, uses the information in it to complete the operation, and then restarts *Op*.

ISB helping goes a long way towards making a data structure recoverable: updates are idempotent and not susceptible to the *ABA problem*, since they must ensure that an update is done exactly once, even if several processes attempt to help it complete. So, if the system crashes while q is executing *Op*, upon recovery, q can re-execute *Op* to completion by either using the information in the Info structure for *Op* (if it is installed) or by starting from scratch.

To support detectability, when q recovers from a crash that occurred while executing *Op*, its recovery code must be able to access the Info structure I of *Op*. This is achieved by allocating, for every process q, a designated persistent *recovery data* variable that stores a reference to I. Furthermore, a recovering process q must be able to figure out whether its failed operation took effect, and if it did, what its response was. To ensure this, a *result* field is added to the Info structure. Process q, and every process helping *Op*, sets the *result* in I before invalidating I from the relevant nodes. Upon recovery, q accesses its last allocated Info structure and uses it to complete its last operation. If the *result* field of the Info structure is set, the operation took effect, and q returns its value. Otherwise, *Op* did not take effect and it can be restarted.

We have applied the ISB-tracking technique described above, to derive a queue, a linked list, a binary search tree, and an exchanger object. The approach can be combined with a technique we call *direct-tracking*, which borrows some ideas from the log-queue [4], to get an elimination stack. More details for these implementations are provided in the full version (http://arxiv.org/abs/1905.13600).

3 EVALUATION

We implemented an ISB-based linked list (ISB) and compared it with two other detectable linked lists. For the first, we applied the *capsules* transformation of [2] to Harris' linked list [5]. To add persistency instructions to capsules, it is proposed in [2] to either use a general durability transformation [6], (which adds the `pwb` and `pfence` instructions after each access to shared memory) or to add them manually; the second requires deep understanding of both capsules and the algorithm to which it is applied. We measured the throughput of both approaches, namely CAPSULES and CAPSULES-OPT. The second competitor is based on the technique we call *direct tracking*, which is not as general as ISB-tracking.

Experimental setting and benchmarks. We used a 40-core machine with 4 Intel(R) Xeon(R) E5-4610 v3 1.7Ghz CPUs with 10 cores each with hyper-threading support and 25MB L3 cache. Code is written in C++ and compiled using g++ (v4.8.5) with O3 optimizations. Each experiment lasts 5 seconds and each data point is the average of 10 experiments. Keys are chosen uniformly at random from the ranges $[1, 500]$. The list is initially populated by performing 250 INSERTS. We present update-intensive (30% finds) and read-intensive (70% finds) benchmarks. As in [2, 4], since the machine does not contain non-volatile memory (NVM), we simulate `pwb` using `clflush` and `psync` using `mfence`, expecting performance overhead close to the real persistency instruction cost in systems supporting NVM.

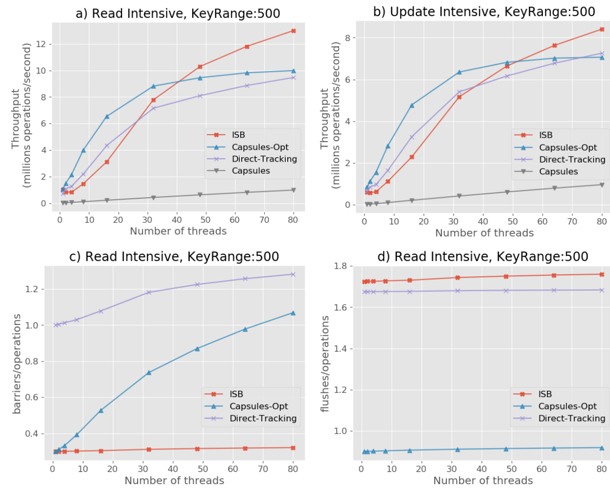

Figure 1: Throughput & numbers of `barriers` and `flushes`.

Experimental Analysis. The results of our experiments are shown in Figure 1. The throughput of the capsules variant that uses the transformation of [6] (CAPSULES) is extremely low due to the prohibitively high number of persistency instructions. As for the rest of the algorithms, it can be seen that ISB's relative performance improves as the number of threads increases. Specifically, in the read-intensive case, the speedup of both DIRECT-TRACKING and CAPSULES-OPT becomes very small after 32 threads, whereas ISB continues to exhibit significant speedup up to the 80 supported threads. ISB's scalability stems from the fact that it performs fewer barriers per operation (see Figure 1c). Specifically, with ISB, a thread has to perform `barrier` (i.e., a `clflush` followed by an `mfence`) only on nodes near its target node, and thus performs a constant number of `barriers` per operation, regardless of the total number of threads. In contrast, both CAPSULES-OPT and DIRECT-TRACKING perform a `barrier` each time they traverse a marked node. As the number of threads increases, `barrier` is performed on more marked nodes, increasing the persistency cost. Notably, neither DIRECT-TRACKING, nor CAPSULES-OPT are as generic as ISB-tracking.

REFERENCES

[1] Hagit Attiya, Ohad Ben-Baruch, and Danny Hendler. 2018. Nesting-Safe Recoverable Linearizability: Modular Constructions for Non-Volatile Memory. In *ACM Symp on Principles of Distributed Computing (PODC)*. 7–16.

[2] Naama Ben-David, Guy E Blelloch, Michal Friedman, and Yuanhao Wei. 2019. Delay-free concurrency on faulty persistent memory. In *31st ACM Symp on Parallelism in Algorithms and Architectures (SPAA)*. 253–264.

[3] 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 *16th International Conf on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[4] Michal Friedman, Maurice Herlihy, Virendra J. Marathe, and Erez Petrank. 2018. A persistent lock-free queue for non-volatile memory. In *23rd ACM SIGPLAN Symp on Principles and Practice of Parallel Programming (PPoPP)*. 28–40.

[5] Timothy L. Harris. 2001. A Pragmatic Implementation of Non-blocking Linked-Lists. In *Distributed Computing, 15th International Conf (DISC)*. 300–314.

[6] Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. 2016. Linearizability of Persistent Memory Objects Under a Full-System-Crash Failure Model. In *30th International Symp on Distributed Computing DISC*. 313–327.

[7] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. 2011. Consistent and Durable Data Structures for Non-Volatile Byte-Addressable Memory. In *9th USENIX Conf on File and Storage Technologies*. 61–75.