Brief Announcement: Proust: A Design Space for Highly-Concurrent Transactional Data Structures

Thomas Dickerson
Brown University
thomas_dickerson@brown.edu

Paul Gazzillo
Yale University
paul.gazzillo@yale.edu

Maurice Herlihy
Brown University
maurice_herlihy@brown.edu

Eric Koskinen
Yale University
eric.koskinen@yale.edu

ABSTRACT

Most STM systems are poorly equipped to support libraries of concurrent data structures. One reason is that they typically detect conflicts by tracking transactions' *read sets* and *write sets*, an approach that often leads to false conflicts. A second is that existing data structures and libraries often need to be rewritten from scratch to support transactional conflict detection and rollback. This brief announcement introduces *Proust*, a framework for the design and implementation of transactional data structures. Proust is designed to maximize reuse of existing well-engineered libraries by providing transactional "wrappers" to make existing thread-safe concurrent data structures transactional. Proustian objects are also integrated with an underlying STM system, allowing them to take advantage of well-engineered STM conflict detection mechanisms. Proust generalizes and unifies prior approaches such as boosting and predication.

1 INTRODUCTION

Software Transactional Memory (STM) has become a popular alternative to conventional synchronization models, both as programming language libraries [7, 10, 15, 18] and as stand-alone systems [1, 3, 8, 13, 16, 17]. STM systems structure code as a sequence of *transactions*, blocks that are executed *atomically*, meaning that steps of concurrent transactions do not appear to interleave.

Most STM systems, however, are not well-equipped to support libraries of concurrent data structures. One limitation is that STM systems typically detect conflicts by tracking transactions' *read sets* and *write sets*, an approach that often leads to false conflicts, when operations that could have correctly executed concurrently are deemed to conflict, causing unnecessary rollbacks and serialization. Instead, it would be preferable to exploit data type semantics to enhance concurrency by recognizing when operations do not interfere at the semantic level, even if they might interfere at some lower level.

A second limitation is that existing thread-safe libraries and data structures must typically be rewritten from scratch to accommodate idiosyncrasies of the underlying STM system. The prospect of discarding so much carefully engineered concurrent software is unappealing. Instead, it would be preferable to provide a pathway for porting at least some existing thread-safe concurrent data structures and algorithms into STM systems.

We are not the first to identify these limitations. *Transactional boosting* [12] describes a methodology for constructing a transactional "wrapper" for prior thread-safe concurrent data structures. A boosting wrapper requires identifying which operations commute, as well as providing operation inverses. Boosting is a stand-alone process, not integrated with an STM. *Transactional predication* [2] describes a way to leverage standard STM functionality to encompass highly-concurrent sets and maps. Predication, however, does not appear to extend beyond sets and maps, and does not provide an explicit path to migrate legacy data structures and libraries. *Software transactional objects* [14] (STO) is an STM design that provides built-in primitives to track conflicts among arbitrary operations, not just read-write conflicts. It does not provide a migration path for legacy libraries.

This brief announcement introduces *Proust*[1], a framework which generalizes key insights from transactional boosting and predication. Proust is designed to ease reuse of existing well-engineered libraries in two ways. First, Proust is a methodology for the design and implementation of transactional "wrappers" that transform existing libraries of thread-safe concurrent data structures into transactional data structures so as to minimize false data conflicts. Unlike predication, Proust supports objects of arbitrary abstract type, not just sets and maps. Second, unlike boosting, Proustian objects are integrated with an underlying STM system, allowing them to take advantage of well-engineered STM conflict detection mechanisms.

Two key elements are necessary to wrap an existing non-transactional concurrent data structure into an STM-compatible object. First, as with boosting, it is necessary to characterize the commutativity relationships between the various operations on that data structure. For example, in a map, queries and updates to non-intersecting key ranges commute. Sometimes, this determination

Supported in part by NSF CCF Awards #1421126 and #003991.

PODC '17, July 25-27, 2017, Washington, DC, USA

ACM ISBN 978-1-4503-4992-5/17/07.
https://doi.org/10.1145/3087801.3087866

[1]This name is a portmanteau of *predication* and *boosting*, both influential prior works. The name is also an *hommage* to Marcel Proust, an author famous for his exploration of the complexities of memory.

can be performed automatically by reduction to SMT[2] solvers. Moreover, in most cases these relationships can be conservatively approximated with traditional two-phase locks, without much loss in expressivity.

Second, the wrapper must provide an *update strategy*, either by providing an inverse for each operation, or a *shadow copy*[3] functionality. Inverses and shadow copies correspond to alternative update strategies, as discussed in Section 2.

This paper makes several contributions. First, techniques for transforming black-box highly-concurrent linearizable data structures into transactional objects with minimal false conflicts, generalizing key insights of boosting and predication. Second, the concept of a *conflict abstraction* that translates an abstract data type's semantic notions of conflict into concrete forms that can be efficiently managed by a generic software transactional memory run-time. Third, systematic guidelines for choosing a transactional API for an underlying thread-safe data structure. Effectively, the transactional API must choose between *optimistic* or *pessimistic* conflict resolution, and *eager* or *lazy* update strategies (Section 2).

In addition to these conceptual contributions, we also developed the ScalaProust prototype implementation, built on top of ScalaSTM. It shows scalability matching existing specialized approaches, but with a much wider range of applicability (Section 3).

2 OVERVIEW

The Proust Design Space The Proust methodology, like Boosting, Predication, and optimistic transactional boosting (OTB) [2, 11, 12] is based on the principles that (1) synchronization conflicts should be defined over an object's abstract (not concrete) state, and (2) the abstract state can be mapped to an underlying STM mechanism by proper synchronization and an ability to roll back changes.

By the Proust *design space*, we mean the following several implementation choices. Concurrency control can be optimistic (as in Predication, OTB) or pessimistic (as in Boosting). The base data structure can be modified *eagerly* as the transaction executes, or *lazily* postponed to commit time. Each prior work cited commits to one fixed choice from each category, while Proust provides a unifying structure allowing choices to be mixed and matched.

The Proust Methodology Proust detects and resolves synchronization conflicts through *conflict abstractions*, which are (roughly speaking) maps carrying abstract states to concurrency control primitives provided by an underlying STM. At the concrete end, programmers are responsible for providing a *lock allocator policy* (LAP), which allocates concurrency control primitives as needed. The LAP is either optimistic or pessimistic. A pessimistic LAP allocates standard re-entrant read-write locks, while an optimistic LAP returns an object, which maps `lock` invocations to operations on standard STM memory locations, allowing the STM to detect and manage synchronization conflicts.

Programmers also choose whether wrapped objects are modified lazily or eagerly. A lazy strategy requires the ability to construct

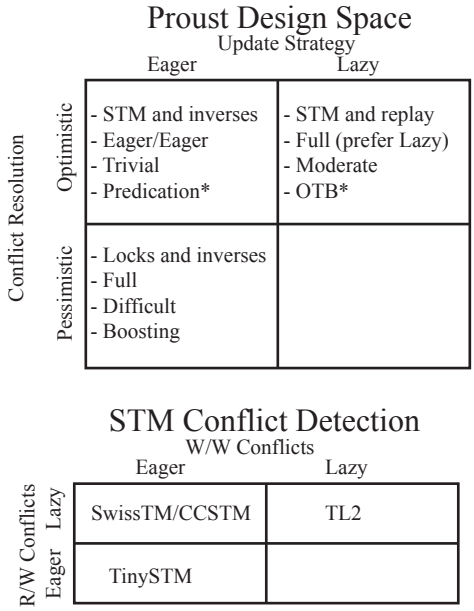

Figure 1: Design spaces for STMs and STM-integrated data structures. The top table outlines the Proust design space, listing for each how the transactional API is implemented, its compatibility with the STM strategies below, the difficulty of correctly synchronizing an AbstractLock implementation with the underlying STM, and the most conceptually similar prior work. The bottom table maps conflict detection strategies to popular STMs as outlined by Dragojevic, et al [6].

a shadow copy, while an eager strategy requires each operation to have a declared inverse, registered as a rollback handler by the abstract lock. If shadow copy functionality is provided, each operation on the wrapped object is forwarded through a *replay log*. The replay log computes the result of the operation at execution time using the shadow copy, and registers a handler to reapply the operation to the wrapped object.

There are many considerations in making these choices, depending on the data structure's operations, or the strengths and weaknesses of the underlying STM system (Figure 1).

Not all combinations make sense. For example, the empty quarter in Figure 1 reflects an impractical combination of choices. Some combinations are more complicated. For example, the eager-optimistic combination satisfies *opacity* [9] only under STMs that provide eager detection of both read-write and write-write conflicts. Here, Proust differs from Predication. While both are eager, Predication delegates state modifications to the underlying STM, instead of using the STM only for synchronization. Some second-order considerations include the degree to which the STM's contention management is exposed and can be coupled with traditional pessimistic locks, and the memory overhead of allocating shadow copies on target systems.

3 EVALUATION

We evaluated several of our map implementations for ScalaProust with a benchmarking setup similar to that used by Bronson, et al. for predication [2]. For each experiment, running on an Amazon

[2] n.b. "Satisfiability modulo theories", **not** "Software Transactional Memory"
[3] A shadow copy essentially provides copy-on-write semantics. The most effective way to provide this functionality is type-dependent.

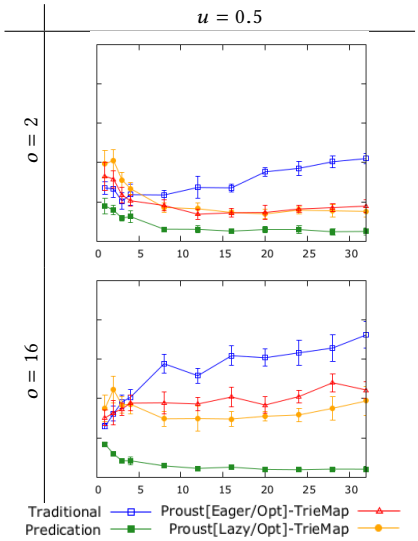

Figure 2: Time to process 10^6 operations on concurrent maps, using a 32-core Amazon EC2 `m4.10xlarge` instance, as the number of threads increases. Each chart is the result for a particular fraction of writes and operations per transaction. For each chart, the x-axis is the number of threads from 0 to 32 and the y-axis is the average time in milliseconds from 0 to 250.

EC2 `m4.10xlarge` instance[4], we performed 10^6 randomly selected operations on a shared map, split across t threads, with o operations per transaction. A u fraction of the operations were writes (evenly split between `put` and `remove`), and the remaining $(1-u)$ were `get`. We varied $1 \leq t \leq 32$, $1 \leq o \leq 256$, and $u \in \{0, 0.25, 0.5, 0.75, 1\}$.

The experimental results depicted in Figure 2 display the effects of several competing trends. Initial results show that, as expected, Proust's performance on map operations parallelizes better than the traditional STM approach, though not as well as the highly engineered predication approach.

The full version of this paper also demonstrates the expressivity of Proust by walking through the implementation of a wrapper for a concurrent priority queue providing an efficient shadow copy operation [4].

4 CONCLUSIONS

We believe that the Proust methodology serves a useful niche in the transactional data structures ecosystem. Like Boosting, we offer sufficient expressivity to wrap arbitrary data structures, but with reduced design complexity (constraints are expressed as commutativity of updates to abstract state elements, rather than as pairwise commutativity rules between operations). Furthermore, our well-characterized design-space permits the use of different synchronization and update strategies to selectively optimize the performance of wrapped data structures for different STMs and different expected work-loads.

The full version of this paper further elaborates on a number of topics treated only briefly here, including techniques for implementing shadow copies, a formalization of conflict abstraction, opacity correctness arguments for Proustian objects, and an example wrapper for concurrent priority queues. [5]

[4]https://aws.amazon.com/blogs/aws/the-new-m4-instance-type-bonus-price-reduction-on-m3-c4/

REFERENCES

[1] A.-R. Adl-Tabatabai, C. Kozyrakis, and B. Saha. Transactional programming in a multi-core environment. In K. A. Yelick and J. M. Mellor-Crummey, editors, *PPoPP*, page 272. ACM, 2007.

[2] N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun. Transactional predication: High-performance concurrent sets and maps for stm. In *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC '10, pages 6–15, New York, NY, USA, 2010. ACM.

[3] L. Dalessandro, M. F. Spear, and M. L. Scott. Norec: streamlining stm by abolishing ownership records. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 67–78, New York, NY, USA, 2010. ACM.

[4] T. D. Dickerson. Fast snapshottable concurrent braun heaps. *arXiv preprint arXiv:1705.06271*, 2017.

[5] T. D. Dickerson, P. Gazzillo, E. Koskinen, and M. Herlihy. Proust: A design space for highly-concurrent transactional data structures. *arXiv preprint arXiv:1702.04866*, 2017.

[6] A. Dragojević, R. Guerraoui, and M. Kapalka. Stretching transactional memory. In *ACM sigplan notices*, volume 44, pages 155–165. ACM, 2009.

[7] D. Goodman, B. Khan, S. Khan, M. Luján, and I. Watson. Software transactional memories for scala. *Journal of Parallel and Distributed Computing*, 73(2):150 – 163, 2013.

[8] J. E. Gottschlich, M. Vachharajani, and J. G. Siek. An efficient software transactional memory using commit-time invalidation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, April 2010.

[9] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming (PPoPP'08)*, pages 175–184, New York, NY, USA, 2008. ACM.

[10] T. Harris, S. Marlow, S. L. P. Jones, and M. Herlihy. Composable memory transactions. *Commun. ACM*, 51(8):91–100, 2008.

[11] A. Hassan, R. Palmieri, and B. Ravindran. Optimistic transactional boosting. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 387–388, New York, NY, USA, 2014. ACM.

[12] M. Herlihy and E. Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 207–216, New York, NY, USA, 2008. ACM.

[13] M. Herlihy, V. Luchangco, M. Moir, and I. William N. Scherer. Software transactional memory for dynamic-sized data structures. In *Proceedings of the symposium on principles of distributed computing*, pages 92–101, New York, NY, USA, 2003. ACM.

[14] N. Herman, J. P. Inala, Y. Huang, L. Tsai, E. Kohler, B. Liskov, and L. Shrira. Type-aware transactions for faster concurrent code. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 31:1–31:16, New York, NY, USA, 2016. ACM.

[15] Intel Corporation. Intel C++ STM Compiler, Prototype Edition. Web. Retrieved from http://software.intel.com/en-us/articles/intel-c-stm-compiler-prototype-edition/, 20 November 2011.

[16] V. J. Marathe and M. Moir. Toward high performance nonblocking software transactional memory. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 227–236, New York, NY, USA, 2008. ACM.

[17] T. Riegel, C. Fetzer, and P. Felber. Time-based transactional memory with scalable time bases. In *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '07, pages 221–228, New York, NY, USA, 2007. ACM.

[18] The Deuce STM Group. Deuce stm - java software transactional memory. web. Retrieved from http://www.deucestm.org/documentation, 20 November 2011.