Abstract

Transactional data structure libraries (TDSL) combine the ease-of-programming of transactions with the high performance and scalability of custom-tailored concurrent data structures. They can be very efficient thanks to their ability to exploit data structure semantics in order to reduce overhead, aborts, and wasted work compared to general-purpose software transactional memory. However, TDSLs were not previously used for complex use-cases involving long transactions and a variety of data structures.

In this work, we boost the performance and usability of a TDSL, allowing it to support complex applications. A key idea is nesting. Nested transactions create checkpoints within a longer transaction, so as to limit the scope of abort, without changing the semantics of the original transaction. We build a Java TDSL with built-in support for nesting in a number of data structures. We conduct a case study of a complex network intrusion detection system that invests a significant amount of work to process each packet. Our study shows that our library outperforms TL2 twofold without nesting, and by up to 16x when nesting is used. Finally, we discuss cross-library nesting, namely dynamic composition of transactions from multiple libraries.

CCS Concepts • Computing methodologies → Concurrent algorithms.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

PPoPP '20, February 22–26, 2020, San Diego, CA, USA
© 2020 Copyright held by the owner/author(s).
https://doi.org/10.1145/3332466.3374514
Algorithm 1 Transaction flow with nesting

1: TXbegin()
2: [Parent code] ▷ On abort – retry parent
3: nTXbegin() ▷ Begin child transaction
4: [Child code] ▷ On abort – retry child or parent
5: nTXend() ▷ On commit – migrate changes to parent
6: [Parent code] ▷ On abort – retry parent
7: TXend() ▷ On commit – apply changes to thread state

1.2 Our Contribution

Transactional nesting. This work pushes the limits of the TDSL concept in an attempt to make it more broadly applicable. Our main contribution is facilitating long transactions via nesting [7]. Nesting allows the programmer to define nested child transactions as self-contained parts of larger parent transactions. This controls the program flow by creating checkpoints; upon abort of a nested child transaction, the checkpoint enables retrying only the child’s part and not the preceding code of the parent. This reduces wasted work, improves performance and reduces energy consumption.

We focus on closed nesting [10], which, in contrast to flat nesting, limits the scope of aborts, and unlike open nesting [8], is generic and does not require semantic constructs. Nesting does not relax consistency or isolation, and ensures that the entire parent transaction is executed atomically.

The flow of nesting is shown in Algorithm 1. When a child commits, its local state is migrated to the parent but is not yet reflected in shared memory. If the child aborts, then the parent transaction is checked for conflicts. And if the parent incurs no conflicts in its part of the code, then only the child transaction retries. Otherwise, the entire transaction does. It is important to note that the semantics provided by the parent transaction are not altered by nesting. Rather, nesting allows programmers to identify parts of the code that are more likely to cause aborts and encapsulate them in child transactions in order to reduce the abort rate of the parent.

Yet nesting induces an overhead which is not always offset by its benefits. We investigate this tradeoff using microbenchmarks. We find that nesting is helpful for highly contended operations that are likely to succeed if retried.

NIDS benchmark. We introduce a new benchmark of a network intrusion detection system (NIDS), which invests a fair amount of work to process each packet. It features a pipelined architecture with long transactions, a variety of data structures, and multiple points of contention. It follows one of the designs suggested in [4] and executes significant computational operations within transactions, making it more realistic than existing IDS benchmarks (e.g., [6]).

Enriching the library. In order to support complex applications like NIDS, and more generally, to increase the usability of TDSLs, we enrich our transactional library with additional data structures – producer-consumer pool, log, and stack – all of which support nesting. The TDSL framework allows us to custom-tailor to each data structure its own concurrency control mechanism. We mix optimism and pessimism (e.g., stack operations are optimistic as long as a child has popped no more than it pushed, and then they become pessimistic), and also fine tune the granularity of locks (e.g., one lock per stack versus one per slot in the producer-consumer pool).

Evaluation. We evaluate our NIDS application. We find that nesting can improve performance by up to 8x. Moreover, nesting improves scalability, reaching peak performance with as many as 40 threads as opposed to 28 without nesting.

Composition. While most of this work considers nesting in the context of a single library, programmers often wish to access data structures from multiple libraries within the same atomic transaction. We discusses dynamic composition of nested transactions from distinct libraries.

A full version of this work is available via [1].

References